The Expressive Power of Query Languages Based on Logic Programming

Mark Reynolds

January 1988

Department of Computing
Imperial College of Science and Technology
University of London
London SW7 2BZ
United Kingdom

Submitted to the University of London in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Abstract

Datalog, or the Horn-Clause query language, is a successful formalisation of the use of pure logic programming to answer queries asked about finite relational structures. However, Datalog is not expressive enough for some practically useful problems as those logic programmers who feel the need to use the non-declarative feature negation as failure probably know. In this thesis we assess the increased expressive power got by adding negation as failure to the pure logic programming query answering approach.

To achieve our aim we first define a logical query language DEF which is an extension of Datalog and has just as nice syntax and semantics as Datalog. We show that DEF is equally as expressive as the most common fixed point query language which contains a lot of useful queries. We then show that DEF expresses exactly those queries which can be implemented using logic programming with negation as failure.

Acknowledgement

I would like to express my gratitude to my supervisor, Professor Dov Gabbay, for his help and encouragement throughout my PhD work.

I would also like to thank Dr Bob Sullivan and Dr Wilfrid Hodges who gave me good advice when I was trying to decide on a satisfactory subject area in which to do research.

All the graduate students who contributed to making the office and department conducive to good research deserve praise. Pablo de la Quintana deserves special mention for his helpful comments on the thesis. Chi Cheong Tam and David Fuller were very helpful when it came to the practicalities of presenting the thesis in a readable form.

I am very grateful to the Association of Commonwealth Universities for supporting my research.

Many thanks are due to the Dickens family for making my life easy in a foreign country.

Of course I would not have been able to start researching for a thesis if it was not for the great help my parents and family gave me in previous years. They also earn my gratitude for the moral support they gave me while I have been studying.

Finally, I would like to thank Sarah for giving me such reliable support and encouragement so that I was able to persevere through the researching and writing up.

Table of Contents

Introduction 7

1. Background 14

1.1 Model Theory 16

1.2 Query Languages 19

1.3 Logic Programming 27

1.4 Negative Information 35

1.5 Deductive Databases 44

1.6 Datalog 49

1.7 Extending Datalog 53

2. Provability 56

2.1 PROV 56

2.2 Provability and Minimal Models 60

2.3 Provability and Theorem-Proving 61

2.4 PROV and USO 63

2.5 Datalog as a sub-language of PROV 65

2.6 Definite Existential Implications 69

2.7 DEF 77

2.8 IMP 80

3. Fixed Points 84

3.1 Least Fixed Points 84

3.2 FO+LFP 87

3.3 Definition of $\overline{LFP}$ 91

3.4 Expressing FO+LFP in $\overline{LFP}$ 97

3.5 Expressing $\overline{LFP}$ in FO+LFP 100

3.6 FO+LFP versus DEF 105

4. Negation as Failure 112

4.1 A query evaluation process for Negation as Failure 112

4.2 Implementing the QEP 121

5. Declarative Meaning 139

5.1 The Synthesis of a Program 139

5.2 Soundness of the QEP with respect to SYNTH 143

5.3 Correctness of the QEP with respect to SYNTH 148

5.4 Completeness of the QEP with respect to SYNTH 161

6. LPNAF and DEF 184

6.1 Implementing DEF 184

6.2 Expressing LPNAF queries in DEF 193

6.3 Summary 198

7. Nesting 200

7.1 The Clausal Language 202

7.2 Negation as Failure versus Negation as not Provable 205

7.3 Nesting 208

7.4 Finite Implicit Definability and Minimal Models 214

7.5 Polynomial Time computable Queries 219

8. Improved Logic Programming 224

8.1 QUILOG 224

8.2 The Query Evaluation Process 226

8.3 Soundness of QEP 235

9. Conclusion 239

9.1 Review of Aims and Motivation 239

9.2 Account of Results 240

9.3 Further Work for the Database Theorist 241

9.4 Further Work for the Logic Programmer 242

Appendix 245

Bibliography 260

Introduction

We consider the relative expressive powers of various methods of answering questions about finite structures. In particular, we investigate what improvement in expressive power can result when query languages based on pure logic programming are extended to languages based on logic programming with negation as failure.

The queries which we will be interested in are questions which can be asked of any finite structure from some relational language and which give answers yes, no or a relation on the domain of the structure. Query languages are just sets of strings of symbols each of which represents a particular query. The most basic query language is FO which contains all the first-order formulas of a given language and has the obvious semantics. FO represents exactly those queries answerable using Codds's Relational Algebra [Cod70]. Unfortunately, it is not strong enough to express some practically useful queries such as the transitive closure query. This has lead to some recent developments of languages based on computing least fixed points. Such examples as FO+LFP [Gur85] have overcome the deficiency in expressibility of FO.

Does logic programming provide any alternative answers? Logic programming techniques can be applied to these problems in the deductive database setting [GMN84a]. Here we describe a situation (possibly incompletely) using a theory and then ask what has to be true of the situation. This is not specifically designed for our problem of querying finite structures yet we can handle some queries by both describing the structure and defining the relation of correct answers to our query in the theory. For example, Q is defined as the conjunction of R and S if we add the sentence

$\forall \bar{x} \; (\; Q(\bar{x}) \longleftrightarrow R(\bar{x}) \wedge S(\bar{x}) \;)$

to a theory completely describing any finite {R,S}-structure. Unfortunately, when

restricted to Horn-clause theories we can neither completely describe the structure nor define (in the usual sense of implicit definition) the query. Thus, in practice, we get a lot of "don't know" answers.

A different method using much the same set up was pioneered in [ChH85]. Instead of trying to define the query inside the theory, we use the theory plus provability to specify a query. To define the conjunction of R and S as above, we add the sentence

$\forall x\ (\ R(x) \wedge S(x) \rightarrow Q(x)\ )$

to a complete description of the {R,S}-structure. The answer to the query is the relation of tuples which are provably in Q i.e. those tuples which are in the interpretation of Q in any expansion of the queried structure to a model of the sentence. This approach, when limited to relational Horn-Clause theories, is formalised in a language, sometimes called DATALOG, which contains queries such as transitive closure which are not in FO.

New Work

Our new work starts with a definition of a powerful generalisation of DATALOG. PROV contains all the queries which can be defined via provability from any relational theory. A PROV query is an ordered pair, written PROV(P;P), of a finite theory P with relation symbols of two types - the old ones from the language of the queried structures plus some new different ones - and one of the new symbols P. The answer to PROV(P;P), asked of any finite structure of the old language, is the relation of tuples which are in the interpretation of P in any expansion of the structure to a model of P. The expressive power of PROV is equivalent to that of the language USO of universally quantified second order formulae - a language containing queries which are almost certainly too hard to answer in a reasonable amount of time (see [Fag74]). PROV is the query language of which all the other of our new languages are sub-languages.

The least powerful sub-language of PROV which we consider is DATALOG itself. It

is based on provability from Horn-Clause theories in which the old symbols do not appear in the heads of clauses. This restriction ensures that only positive information from the queried structure is needed in answering queries. We consider three possible ways of making more expressive languages:

1) we could allow disjunctions in the heads of clauses - producing indefinite theories ;

2) we could allow negations of old relations and inequality to appear in the bodies of clauses - producing non-positive theories ; or

3) we could allow universal quantification to appear in the bodies of clauses - producing non-existential theories.

By allowing different combinations of these features we get several query languages of differing expressive power and ease of implementation.

The strongest language, IMP, which is based on sentences allowing any of the extensions, is shown to be as powerful as PROV itself. By a result in [Fag74], it can thus express all the negations of non-deterministically polynomial time computable queries. Polynomial time computable queries are thus well within its power - for example , "is the size of the structure an odd number?" can be answered by trying to prove that Odd follows from a complete description of the structure plus the following theory in which R, S and Odd are new symbols:

$\{ \forall xy\ (\ R(x,y) \rightarrow R(y,x)\ ),$

$\forall x\ (\ R(x,x) \rightarrow Odd\ ),$

$\forall xyz\ (\ R(x,y) \wedge R(x,z) \wedge y \neq z \rightarrow Odd,$

$\forall xy\ R(x,y) \vee S(x,y),$

$\forall x\ (\ \forall y\ S(x,y) \rightarrow Odd\ )\}.$

DEF and FO+LFP

Our most important generalisation of DATALOG is the query language DEF based on sentences which can contain old negations, inequalities and universal quantifications as conjuncts in the body but one and only one atomic formula appears as the head and it is built from a new relation symbol. For example, if your brother is liked by everyone then you are a correct answer to the DEF query PROV(P;Q) where P is as follows:

$\{ \forall xyz [parent(z,x) \wedge parent(z,y) \wedge x \neq y \wedge male(y) \wedge (\forall u\ likes(u,y))$

$\rightarrow Q(x)] \}.$

In order to compare DEF with established query languages such as the least fixed point language FO+LFP defined in [Gur85], we note the least fixed point character of logic-programming type deductions. Basing our ideas on the use of a fixed point operator in [EmK76], we invent a new least fixed point language $\overline{LFP}$. It is characterised by the use of one multiple operator defined by several first-order formulae. We show DEF$\equiv\overline{LFP}$, i.e. that DEF and $\overline{LFP}$ are equivalent in expressive power.

By showing that FO+LFP$\equiv\overline{LFP}$, we can conclude that DEF$\equiv$FO+LFP. To show the former we use a result in [Imm82] implying that the nesting of fixed point operators and first-order constructions within each other can actually be collapsed. We must also show that the multiple character of $\overline{LFP}$ can be mimicked by multiple uses of simple LFP operators. This is not a trivial matter of coding the tuples of relations involved in a $\overline{LFP}$ operation into one big relation. Nevertheless, we can conclude that DEF, FO+LFP and $\overline{LFP}$ are all equally expressible.

DEF and NAF

The major results in this thesis are those demonstrating the close relationship between the class of queries implementable by logic programming with negation as failure (LPNAF) and the class of queries expressible in DEF. By a query implementable with

LPNAF, we mean one that can be answered by extending the usual logic programming technique for answering DATALOG queries to the LPNAF technology -i.e. we have an LPNAF program PROG and a symbol P such that finding out whether σ is a correct answer to a query or not can be accomplished by asking "$P(\sigma)$?" of the two-part program consisting of a description of D followed by PROG.

The first problem is to get a workable definition of NAF. On the basis of comments in [She84], we reject any ideas that the closed world assumption or the Clark completed database provide any account of a declarative meaning of an LPNAF program. Instead, we must base our work on a definition of a query evaluation process for NAF. We choose a slightly more complete process (i.e. it gets more answers) than Clark's [Cla78] and so have to relate it back to Clark's.

It would be easier if we could use some sort of declarative meaning of NAF and, in fact, we do so. We show how to construct from each functionless LPNAF program, PROG, a first-order theory, SYNTH(PROG), such that :

1) if $P(\sigma)$ succeeds from PROG then $P(\sigma)$ is deducible from SYNTH(PROG); and

2) if $P(\sigma)$ finitely fails from PROG then $P(\sigma)$ is not deducible from SYNTH(PROG).

Furthermore, we detail some conditions, and say that a program which satisfies them is covered. If PROG is covered then

3) if $P(\sigma)$ is deducible from SYNTH(PROG) then $P(\sigma)$ succeeds from PROG.

By using the results 1 and 2, we get a correctness result - for each functionless and constantless LPNAF program, there is a DEF query such that the answers which are obtained using the program are correct answers to the DEF query. Thus, all queries implemented by LPNAF are in DEF.

By using result 3, we can get a completeness result - for any DEF query, there is an LPNAF program which can be used to answer the query. The program will get all the

"yes" answers for the query, and if the program gets an answer then the answer will be the correct answer for the DEF query.

Stratified Databases

In [ChH85], DATALOG is extended to a query language, **C**, which is based on a hierarchy of sub-programs determining whether negations of new symbols can appear in the bodies of (what would otherwise be) Horn-clauses. Universal quantification is not needed for it is proven in [ChH85] that **C**≡FO+LFP. So we can conclude that **C**≡DEF. The stratified database approach of [ABW86] is much the same as **C**.

The clausal language **C** is not a sub-language of PROV. Although **C** is based on the relationship between a theory and one of its symbols, the relationship is not provability. To define the semantics of **C**, Chandra and Harel nested provability in complementation and complementations in provability. It is suggested in [ChH85] that NAF is relevant to implementing this language. We show that this is not so because finite failure to prove is not the same as not provable. Implementing **C** involves knowing what is not provable in a sub-program in order to be able to compute negations in a higher (in the hierarchy) sub-program.

In order to implement a stratified database using LPNAF one must therefore use the result in [Imm82] converting nestings of negations and fixed point operators to an alternative single unnested fixpoint. Thus, we are essentially compiling the stratified database into a DEF theory before attempting to answer the query. We make some brief investigations into what other provability languages are similarly closed under nestings of complementations. The idea of finite implicit definability is relevant here : we show that all polynomial time computable queries are implicitly definable on finite structures, and so nesting them under complementation does not offer any increased expressibility over straight provability semantics.

Implementations

The relationship between NAF and DEF suggests an alternative approach to implementing DEF queries. Perhaps we could program directly with DEF theories. It would be a purely logical approach (unlike NAF) and yet suffer no loss of expressibility compared to LP with NAF nor any efficiency (this is because of the close relationship between doing deductions in SYNTH(PROG) with programming with PROG). We develop an improved unification algorithm supporting this development. It involves inequalities as well as unifications and can handle problems like the following (which simple approaches such as Skolemising out universal quantifications have trouble with):

$\forall y\ [\ (\ \forall x\ L(x,y)\ )\ \rightarrow\ B(y)\ ]$,

L(M,M)

L(J,M) (*)

$\forall x\ [\ x \neq M \wedge x \neq J\ \rightarrow\ L(x,M)\ ]$.

From this theory B(M) is concluded but the theory without the starred sentence fails to lead to B(M).

Chapter 1

Background

What are query languages? What is the expressive power of a query language? What is logic programming? How can a query language be based on logic programming? Why study the expressive power of query languages based on logic programming? In this chapter, we shall endeavour to answer these questions.

These concepts and problems arise in the overlap between the study of databases and the study of logic programming. While these are two distinct areas of research, they are both ultimately concerned with the general task of answering complex questions about complex situations - the core of database research perhaps being more concerned with the logical representation of a particular type of situation (i.e. finite relational structures) and the core of logic programming research looking more at a certain way of computing answers once a logical specification is given.

Both these areas and this thesis are based upon fundamental ideas from mathematical logic. In order to get started, we first have a quick look at these fundamental ideas - getting some terminology straight in the process. The three most basic ideas are those of structures, theories and models. A structure is a formalisation of the idea of a situation. A theory formalises the idea of a description which may or may not apply to a given situation. If the description is actually appropriate to the situation then we say that the structure is a model of the theory.

Green in [Gre69] and Kuhns in [Kuh67] and [Kuh70] both used logic for aspects of

knowledge representation but the logical foundations for database theory were laid by Codd in [Cod70]. He took the idea of a finite relational structure from mathematical logic as the basis of his relational data model.

In section 1.2, we will see how Codd's formalisation of the representation of a complex situation allows us to then go on to formalise the idea of a question asked about a situation. We shall introduce queries and then query languages. Query languages represent whole classes of queries and we can ask when one language has more expressive power than another - i.e. when does the former represent more queries than the latter. We shall see how the focus of research has moved from the weaker first-order query languages to the more powerful fixed point languages.

In section 1.3 we shall switch to concentrate on logic programming. We will see to what extent logic programming lives up to the motivation for a declarative programming language - i.e. a language using which a program describes the answers to the problem which it solves as well as a method of solving the problem.

Pure logic programming has some serious limitations when it comes to specifying problems about real situations. To logic programmers, one of the most annoying is the inability to include rules involving negative information. Unfortunately, handling negative information introduces formidable computational problems and the outcome of attempts to come to some sort of compromise have been unsatisfactory. In section 1.4 we shall see how logic programming has been extended in this way to include the computational rule, negation as failure.

In section 1.5 we see how logic programs have been used in database problems. The kinds of problems that such approaches can handle are so different to those described in section 1.2 that we identify two distinct schools of database research - conventional database theory as in 1.2 and deductive database theory as in 1.5. However, we shall see

that deductive database theory is based on some shaky foundations.

In section 1.6 we will see how we can take ideas from deductive database research that are both helpful to our task of asking questions of finite structures and based on safe foundations. We end up with a query language, Datalog, based on logic programming.

Finally, in section 1.7, we will identify a need - the lack of expressive power of Datalog concerning some practical problems - and a means - the possibility of extending one of the ways of evaluating Datalog queries from using pure logic programming to using logic programming with negation as failure. We identify some theoretical questions which need answering before a query language implementable using logic programming with negation as failure can be hailed as practically useful.

Hopefully, after reading this chapter, the reader will understand what we are setting out to do and why.

1.1 Model Theory

In this section we will quickly go over the very basic basics of model theory. All the details appear in the beginning of [ChK73] and we will just mention concepts which might be different, or appear in different notation, in other treatments of the basics of model theory and mathematical logic.

In the context of model-theoretic discussions in this thesis a *language* (commonly known elsewhere also as a signature or a vocabulary) is a set of symbols of three sorts - *relation symbols*, *constant symbols* and *function symbols*. Each relation symbol is associated with a non-negative integer called its *arity* and each function symbol is associated with a positive integer called its *arity*. A symbol of arity n is said to be *n-ary*.

We shall pay particular attention to finite languages consisting wholly of relation symbols. We shall reserve the term *signature* for such languages.

If **L** is a language then an **L**-*structure* is an ordered pair (S,α) consisting of a non-empty set S and a map α which takes each n-ary relation symbol in **L** to an n-ary relation on S, each n-ary function symbol to an n-ary function on S and each constant symbol in **L** to an element of S. S is called the *domain* of the structure and the image of each symbol from **L** under α is called the *interpretation* of the symbol. Just in case there is any doubt about exactly what a relation on a set is, we will consider 0-ary relations as either being truth (i.e the truth value of a tautology) or falsity and n-ary relations, for positive n, as being a set of n-tuples from the set.

We will be particularly interested in structures with finite domains which we will call *finite structures*.

Many times in this thesis we will be taking an **L**-structure and defining interpretations for extra symbols which are not in **L**. We shall call this *expanding* the structure - if $\mathbf{L}\subseteq\mathbf{K}$ are both languages and D is an **L**-structure then we will call a **K**-structure E an *expansion* of D if and only if the domains of D and E are the same and the interpretations for all the symbols from **L** are the same in E as in D.

The opposite of expansion is *reduction*. If $\mathbf{L}\subseteq\mathbf{K}$ are languages and E is a **K**-structure expanding the **L**-structure D then we will call D the **L**-*reduct* of E - each **K**-structure E has only one **L**-reduct D and we write this as $E|_L$.

If **L** is a language then, in the usual way, we form the first-order language of **L** with identity consisting of **L**-*terms*, *atomic* **L**-*formulae*, **L**-*formulae*, and **L**-*sentences*. We also define the ideas of *free* and *bound* occurrences of variables in formulae.

If ϕ is an **L**-formula and $\vec{x}=(x_1,\cdots,x_k)$ is a tuple of distinct variable symbols

then we write $\phi=\phi(x)$ if and only if all the free variables of ϕ appear amongst the x_i. If we have introduced ϕ by writing $\phi=\phi(x)$ as above and r is a k-tuple of L-terms then $\phi(r)$ is the result of substituting each free occurrence of x_i in ϕ by t_i. For formulas ϕ which have not necessarily been introduced as $\phi=\phi(x)$ we write $\phi(x\backslash r)$ for ϕ under the same substitution.

Such substitutions are also allowed to act on tuples of terms in the obvious way. For example, if α contains variables from y then $\alpha(y\backslash b)$ is α with each appearance of y_i replaced by b_i, while if α has been introduced as $\alpha=\alpha(x)$, then $\alpha(a)$ is $\alpha(x\backslash a)$.

If ϕ contains no free variables, i.e. ϕ is a sentence, we shall sometimes say that ϕ is *closed* and in other circumstances we shall say that ϕ is *ground*.

A set of L-sentences is called an L-*theory*.

If D is an L-structure, $\phi=\phi(x)$ an L-formula, $x=(x_1, \cdots ,x_k)$, $\bar{d}$ a k-tuple of elements from D then we specify when D is a *model* of $\phi(\bar{d})$, $D \models \phi(\bar{d})$, in the usual way (i.e. by induction on the construction of ϕ). For example, $D \models a=b$ if and only if a and b are the same element of the domain of D.

We will find that a concept used in such definitions will be useful for us in this thesis. If L is a language and M an L-structure then we will call a map α from L-terms to elements of M an M-*substitution* if and only if it satisfies

MS1) if a is a constant from L then $a\alpha$ is the interpretation of a in M and

MS2) if f is a function from L and $\bar{e}=(e_1, \cdots ,e_k)$ a k-tuple of L-terms then $f(\bar{e})\alpha$ is the the image of $(e_1\alpha, \cdots ,e_k\alpha)$ under the map which is the interpretation of f in M.

We allow M-substitutions to operate on tuples of L-terms giving tuples of elements from M in the obvious way.

We say D is a *model* of a theory T if and only if D is a model of all the sentences in

T.

We assume that the reader is familiar with the idea of an L-sentence ϕ being *deducible* from an L-theory T. We write $T \vdash \phi$ if this holds. It must be emphasised that we are doing deductions in the first-order language of L with identity so the identity axioms are available for use in such deductions.

We conclude by stating a theorem which we will make great use of in this thesis - always implicitly.

THEOREM 1.1 (easy consequence of [ChK73] theorem 1.3.21)

ϕ is deducible from T if and only if every model of T is also a model of ϕ.

1.2 Query Languages

In this section we will have a quick look at the conventional database formalisation for the problem of asking questions of complex situations. More details can be found in [Del80] and [Mai83].

In [Cod70] Codd laid the foundations for this approach by suggesting that information about a situation should be stored in a database in a computer that can be interacted with as if it was a finite relational structure. For example, if D is the structure, R an n-ary relation symbol from the signature, $\bar{a}$ an n-tuple from the domain of D then the user should find it very easy to find out whether $D \models R(\bar{a})$ or not. Codd actually allowed a typed structure - i.e. different domains for different coordinates of a relation - but, along with most of the researchers which will be mentioned subsequently, we will get by without that complication. [Dat77] , [Dat81] and [Ull82] contain more recent presentations of this *relational* (or *conventional*) approach to databases.

Now that we know what we are asking questions about, let us replace the vague idea of a question by the formal concept of a query. A query will be something which picks out some tuples in the domain of a structure as being correct answers. To define a query formally we will use a variant of the concept of a global relation as introduced by Gurevich in [Gur83] and set out in [Gur85].

If **L** is a signature and n is a non-negative integer then an n-ary **L**-*query* is a map which takes each finite **L**-structure D to an n-ary relation on the domain of D. If q is an n-ary **L**-query and D an **L**-structure then we say that an n-tuple $\bar{a} \in D^n$ is a *correct answer* to q if and only if $\bar{a} \in q(D)$.

A specific example follows. If we know the immediate bosses of employees in a large hierarchically organised company then we might ask about who is ultimately superior to whom. Here **L** could contain only a 2-ary symbol boss_of and given an **L**-structure representing the organisation in the obvious way the correct answers to our query are the ordered pairs of employees in the ordered transitive closure of the boss_of relation. Superiority is a 2-ary {boss_of}-query.

The simplest, more general, examples of **L**-queries are first-order **L**-formulae. Suppose $\bar{x}=(x_1, \ldots, x_n)$ and $\phi(\bar{x})$ is a first-order **L**-formula. Then ϕ represents the n-ary **L**-query q defined as follows:
given an **L**-structure D, the n-ary relation q(D) on the domain of D is just the set of n-tuples which satisfy ϕ in D

$$\text{i.e.} \quad \forall\, \bar{a} \in D^n, \quad \bar{a} \in q(D) \iff D \models \phi(\bar{a}).$$

Technically, it is a first-order formula along with an ordering on its free variables (plus maybe other variables) which determines a query. For example, if ϕ is the formula
$P(x,y) \wedge Q(y,u)$
and $D \models P(a,b) \wedge Q(b,c)$, then a query which we introduce as being represented by

$\phi(x,y,u)$ has (a , b , c) as a correct answer, while $\phi(u,x,y,z)$ represents a query with (c , a , b , d) as a correct answer.

We have seen how first-order L-formulae can represent L-queries. This is a specific case of a query language. If L is a signature then a *query language* on L is a class of strings of symbols (called *well formed formulae, wff*, of the language) each of which represents an L-query in some specified way. We will denote the query language consisting of first-order formulae on L, with the method of representation as defined above, as FO(L).

We shall generalise the model-theoretic notation of satisfiability to be useful for all query languages.

If σ is a wff from a query language on L and represents the n-ary L-query q

then for all L-structures D , for all $\bar{a} \in D^n$,

we will write $D \models \sigma(\bar{a})$ and say that $\bar{a}$ is a *correct answer* to σ in D

if and only if $\bar{a} \in q(D)$.

This is a generalisation, since for $\phi \in FO(L)$,

$D \models \phi(\bar{a})$ *in the model-theoretic sense*

$\Leftrightarrow D \models \phi(\bar{a})$ *in the query sense.*

Codd, in [Cod72], developed the first query languages for his relational model [Cod70]. He introduced the relational calculus and the relational algebra and showed that they had the same expressive power. He proposed that they were to be the basis of a notion of completeness for all query languages: a query language was to be called complete if it could express all the queries definable by the relational calculus.

Bancilhon [Ban78] and Paredaens[Par78] both have shown that Codd's languages can express all first-order definable relations i.e. the relational calculus has the same expressive power as FO. Codd's notion of completeness is just the same as being as

expressive as the first-order query language.

However, there are languages much more powerful than FO and for this reason, Codd's definition of completeness is not totally satisfactory. Two related, very powerful languages are the universal and existential second-order languages, USO and ESO respectively. The wff of USO(**L**) are those of the form $\forall S\phi(x)$ where **S** is a signature disjoint from **L** and ϕ a first-order formula from **L**$\cup$**S**. The wff $\forall S\phi(x)$, with $x=(x_1, \ldots, x_n)$, represents the n-ary **L**-query q, where, for all **L**-structures D, for all $a \in D^n$,

$$a \in q(D) \iff D \models \forall S\phi(a) \iff \forall \text{ } \mathbf{L}\cup\mathbf{S}\text{-structures } E \text{ such that } E|_{\mathbf{L}}=D, \; E \models \phi(a).$$

ESO(**L**), on the other hand, has formulas $\exists S\phi(x)$ such that
$D \models \exists S\phi(a)$ if and only if there is an **L**$\cup$**S**-structure E such that $E|_{\mathbf{L}}=D$ and $E \models \phi(a)$.
The negation of each ESO query is in USO and vice-versa.

If **L**=$\emptyset$, then an example of a 0-ary ESO(**L**) query is $\exists S\sigma$, where **S** has 2-ary R, and σ is as follows:

$$\forall x \; [\; \neg R(x,x) \;] \wedge \; \forall xy \; [\; R(x,y) \rightarrow R(y,x) \;] \wedge$$
$$\forall xyz \; [\; R(x,y) \wedge R(x,z) \rightarrow y=z \;] \wedge \; \forall x \; \exists y \; R(x,y).$$

It should be apparent that, for all finite **L**-structures D, $D \models \exists S\sigma$ if and only if the size of D is even.

A result of Fagin's demonstrated that ESO is not a very practical query language from a computing point of view. In [Fag74], he showed that the 0-ary ESO queries are precisely those 0-ary queries which can be computed by a non-deterministic machine in polynomial time. Given the current feelings that $P \neq NP$ (see for example [DaW83]) are correct some ESO queries would take a long time to compute.

However, there are equally practical reasons why its no good just sticking to FO. In [AhU79], Aho and Ullman point out that the relational calculus (and hence FO) can not

express the transitive closure of a given relation and yet this is a widely used query. Hajek [Haj75] actually gave the first proof that a transitively defined property- in this case connectivity of finite graphs - is not first-order definable. Fagin [Fag75] gave a more general theorem implying the same result. Since then Gaifman [Gai82] has given a general theorem on the limitations of first-order expressibility but the simplest such proof is in [GaV83].

One way to overcome this shortcoming is to add a transitive closure operator directly to the query language (as suggested in [Zlo79]). Immerman [Imm83] explores this possibility and some of the interesting results are given in [Gur85].

However, the most developed extensions of FO are the least fixed point languages. Early work in utilising this idea includes [AhU79], [ChH80a] , [ChH80b], [Imm82], [Var82], [Liv83] and [Gur83].

The foundations for such languages were laid long ago by Tarski [Tar52]. If X is a partially ordered set then we say that X is *complete* if every subset of X has a least upper bound and a greatest lower bound. We say that a map F:X→X, i.e. an operator on X, is *monotone* if and only if

$$\forall x,y \in X \;\; x \leqslant y \;\; \Rightarrow \;\; F(x) \leqslant F(y),$$

and we call $x \in X$ a *fixed point* of F if and only if F(x)=x. Tarski proved that if X is a complete partially ordered set, then every monotone operator on X has a *least fixed point* i.e. a fixed point $x \in X$ such that if $y \in X$ is also a fixed point of F then $x \leqslant y$.

If **L** is a signature and D is a finite **L**-structure then the set $REL_n(D)$ of all n-ary relations on the domain of D is a partially ordered set with the following order $\leqslant$:
If $\sigma,\rho \in REL_n(\mathrm{D})$ then $\sigma \leqslant \rho$ if and only if $\forall \bar{a} \in D^n \;\; \bar{a} \in \sigma \;\; \Rightarrow \;\; \bar{a} \in \rho$.

Since $REL_n(D)$ is finite, $REL_n(D)$ is complete.

Now suppose that P is an n-ary relation symbol not in L, $\bar{x}=(x_1, \ldots, x_n)$ and $\phi(\bar{x})$ is an $L\cup\{P\}$-formula. We will make an operator F_ϕ on $REL_n(D)$ out of ϕ. Given $\sigma\in REL_n(D)$ we must define an n-ary relation $F_\phi(\sigma)\in REL_n(D)$. We proceed as follows:

$$\forall \bar{a}\in D^n,\quad \bar{a}\in F_\phi(\sigma) \iff D\models\phi(P\backslash\sigma,\bar{x}\backslash\bar{a}).$$

The notation $D\models\phi(P\backslash\sigma,\bar{x}\backslash\bar{a})$ means that $\phi(\bar{a})$ is true in the $L\cup\{P\}$-structure with L-reduct D and the n-ary symbol P interpreted as the n-ary relation σ.

For example, L might have the 1-ary symbol "Cat" in it. We can define an operator F_ϕ on the 2-ary relations of an L-structure via the formula

$\phi(x,y) = (\ Cat(x) \wedge Cat(y) \wedge P(x,y)\ ) \vee P(y,x)$.

If D is the L-structure with domain { Nick, Jane, Fluff, Rover, Tabby } such that Fluff and Tabby are the only Cats then F_ϕ operates as follows:

$$F_\phi(\varnothing)=\varnothing$$

$$F_\phi\left\{\left\{(Nick,Jane),\ (Fluff,Rover)\right\}\right\} = \left\{\left\{(Jane,Nick),\ (Rover,Fluff)\right\}\right\}$$

$$F_\phi\left\{\left\{(Nick,Fluff),\ (Fluff,Fluff),\ (Fluff,Tabby)\right\}\right\}$$
$$= \left\{(Fluff,Fluff),\ (Fluff,Tabby),\ (Fluff,Nick),\ (Tabby,Fluff)\right\}$$

Now, provided that ϕ is a formula of $L\cup\{P\}$ such that for all L-structures D the operator F_ϕ is monotone on $REL_n(D)$ then we could write $LFP[\phi](\bar{x})$ to represent the following query:

$\forall L\text{-}structures D,\ \forall \bar{a}\in D^n,\ D\models LFP[\phi](\bar{a})$ if and only if $\bar{a}\in\rho$

where $\rho\in REL_n(D)$ is the n-ary relation which is the least fixed point of the operator F_ϕ

on $REL_n(D)$. We know that the least fixed point ρ exists by Tarski's theorem since F_ϕ is monotone on $REL_n(D)$.

For example, if L has 2-ary "parent" then we could let

$$\phi(x,y) = parent(x,y) \vee \exists z[\ P(x,z) \wedge parent(z,y)\ ].$$

A little work should convince the reader that the query $LFP[\phi](x,y)$ has correct answers all those (a,b) in an L-structure D such that a is an ancestor of b.

The actual notation which has been developed is a bit more complicated than this simplified version for it allows nested LFP operators. We will need to define the full version in a later chapter. The interested reader is also referred to [Gur84] or [Gur85] for details.

The language FO+LFP, FO with the addition of this form of least fixed point operator, is easily shown to include transitive closure queries (our ancestor query was such a one). It is also not too complicated from a computational point of view. In fact one major reason for research into the LFP language is its close relationship with the class P of polynomial time computable queries. In [ChH80b] it is shown that all FO+LFP queries are polynomial time computable but there exist queries in P which are not in FO+LFP. That the domain has an even size is one query in P but not in FO+LFP. Similar results are also proven in [BGK84]. Surprisingly, Immerman [Imm83] has shown that there are polynomial time complete problems in FO+LFP. Immerman [Imm82] and Vardi [Var82] have shown that if we restrict our attention to totally ordered structures then the P queries we can ask on them are all expressible as FO+LFP queries (and vice versa).

In the simplified definition of the LFP operator which we gave we specified that, for $LFP[\phi]$ to be defined, ϕ, a formula of $L \cup \{P\}$, should be such that F_ϕ is monotone on REL_n(D) for all L-structures D. A theorem of Lyndon's, in [Lyn59], says that if F_ϕ is monotone on REL_n(D) for all, finite or infinite, L-structures D, then ϕ is logically

equivalent to an $L \cup \{P\}$-formula ψ which is *positive* in P i.e. P does not appear nested in an odd number of negations in ψ. For this reason, the actual language FO+LFP used by most researchers allows LFP to operate only on formulas positive in P.

However, to use Tarski's theorem to guarantee the existence of a least fixed point of F_ϕ for our applications we only need F_ϕ to be monotone on all finite structures. This suggests a more powerful language FO+LFP'. The wff of FO+LFP (L) can have only LFP[ϕ] for ϕ positive in P - i.e. F_ϕ monotone on the n-ary relations from all L-structures - , the wff of FO+LFP' (L) can have LFP[ϕ] for F_ϕ monotone only on the n-ary relations of all finite L-structures. Ajtai and Gurevich [AjG85] show that Lyndon's theorem doesn't work when we restrict to finite structures only - i.e. there are some ϕ such that F_ϕ is monotone on the n-ary relations of all finite L-structures but ϕ is not logically equivalent to a formula positive in P.

Gurevich and Shelah [GuS85] investigate the relative powers of FO+LFP, FO+LFP' and another related language FO+IFP (using the so called non-monotonic or inductive fixed point). In actual fact, each of these three languages is as expressible as the others. Although FO+LFP' uses wff based on some ϕ unavailable to FO+LFP, the fixed points of these turn out to be expressible as fixed points in FO+LFP.

We have noted that the full definition of FO+LFP allows nesting of LFP operators. In fact, as shown by Immerman [Imm82] only one level of nesting would suffice to express all the FO+LFP queries. A similar result for FO+IFP is proven in [BoG86] .

Gaifman, in [Gai84], developed a generalisation of FO+LFP which is based on operators on tuples of relations i.e. on cartesian products of $REL_{n_i}(D)$. We shall employ a simple version of this query language to show the close relationship of fixed point languages with aspects from the theory of logic programming.

Before we look at logic programming we shall define some notation to help us relate query languages. If A(L) and B(L) are query languages on a signature L then we will write $A(L) \subseteq B(L)$ if and only if every query represented by a wff from A(L) is also represented by a wff from B(L). We write $A(L) \equiv B(L)$ and say that A(L) and B(L) are equally expressive if and only if both $A(L) \subseteq B(L)$ and $B(L) \subseteq A(L)$. If, for every signature L, A(L) and B(L) are query languages which are equally expressive (respectively $A(L) \subseteq B(L)$), then we will write $A \equiv B$ (respectively $A \subseteq B$).

1.3 Logic Programming

What is logic programming? In this section we will look at how the terms declarative programming, logic programming and PROLOG are used and say how they will be used in this thesis. We will also describe, in some detail, one way in which a computer can work on a problem expressed by a logic program.

Programming is the activity of writing instructions which tell computers what to do. Specific relationships between the written symbols and the computers actions (i.e. changes of internal states) are summed up in specific *programming languages*. A *program* is a set of instructions, expressed in a programming language, for the solution of a particular problem. Declarative programming languages, logic programming languages and PROLOG are three terms signifying three overlapping and not necessarily well defined classes of programming languages.

We start by considering the *procedural approach* to a problem. This is simply describing an algorithm for the solution of the problem. Writing a program is, by its nature, approaching a problem in a procedural way - although we reserve the term procedural programming language for members of a certain class of languages. The

procedural interpretation of a programming language is the way in which the computer acts according to the instructions of the language.

Contrasted to the procedural approach is the *declarative approach* to a problem. This involves describing (in some language) properties of a correct answer to the problem and forgetting about how such a solution is to be found. A *declarative interpretation* of a programming language is some description of the answers which the computer arrives at after running programs written in that language- it does not have to mention the details of the algorithm that was used.

The difference between the procedural approach and the declarative approach to problems is at the heart of a lot of the issues of ease of understanding of programs and proof of correctness of programs. From these points of view it would be nice if a program could convey explicitly a lot of declarative information about the problem it solves. However, it might not seem possible that a program could be both a description of the problem and instructions on how to solve it.

A *declarative* programming language is one that overcomes this apparent obstacle by allowing its programs to be read by us as descriptions of the answers to a problem while, at the same time, being able to be read by the computer as instructions for the solution of the same problem. There are general discussions of this idea in [BYT85].

Two classes of declarative programming languages are popularly identified - functional languages and logic programming (or relational) languages. Functional languages [Dar84] are characterised by their declarative interpretations as defintions of functions (often recursive). Their origins can be traced to work on the lambda calculus and recursion equation systems ([Chu41] and [Kle36]).

We are interested in the logic programming languages. These have been developed since Kowalski, in [Kow74], showed that the language of logic, a particularly good

language for describing problems, could, in some circumstances, also have a procedural interpretation.

Up until that time, programming and doing automated theorem proving were different activities. Now the distinction is less clear cut but there still is a difference in emphasis which is worth pointing out. Automated theorem proving is concerned with designing algorithms which can decide whether a given sentence is deducible from a given set of sentences. To some extent the sets of sentences given to a computer using such an algorithm can be regarded as a declarative program - the sentences are a description of the problem and they are read by the computer as the instruction to decide, using its algorithm, whether the given sentence follows from the given set. However, the term declarative program would more often be used to denote a more detailed (lower level) control of the operation of the computer by the programmer.

Kowalski, in [Kow74], and Colmerauer et al, in [CKRP73], first presented the basis for logic programming. Their idea was to have programs which looked like sets of Horn clauses - a Horn clause is a sentence of the form

$\forall x\ (\ \wedge_{i=1}^{q} P_i(x_i) \rightarrow R(y)\ )$

(or equivalently the universal quantification of a disjunction of an atomic formula and a disjunction of negated atomic formulae).

Programs are built from program clauses which are built from *literals*. In order to allow for a generalisation later in the thesis, we shall call these positive literals. A *positive literal* is an atomic formula "R(x)" where R is an n-ary relation symbol and x an n-tuple of terms.

A *program clause* is anything of the form $A \leftarrow A_1, \cdots, A_q$ where A is a positive literal, q is a non-negative integer and each A_i is a positive literal. The head of the clause is A and $A_1, \ldots, A_q$ is its body which may be empty.

If **L** is a language then a *program* from **L** is a finite list of program clauses : each one containing only symbols from **L**.

In the context of logic programming, a *query* is anything of the form $L_1, \cdots, L_p$ where p is a positive integer and each L_i is a positive literal.

It is easy to specify what we want to achieve. The program *represents* a Horn-clause theory **P** in a straight forward way - the program clause $A \leftarrow A_1, \cdots, A_q$ containing distinct variables x represents the Horn-Clause $\forall x (\bigwedge_{i=1}^{n} A_i \rightarrow A)$. The query *represents* a formula ϕ in a straight forward way - $L_1, \cdots, L_p$ with variables x represents $\phi(x) = \bigwedge_{i=1}^{p} L_i$. We want the computer to find a such that $\mathbf{P} \vdash \phi(a)$.

For example, the program

R(y) ← P(x,y), Q(x)

P(x,x) ←

Q(a) ←

represents the Horn-clause theory

$\{ \forall xy (P(x,y) \wedge Q(y) \rightarrow R(y)) , \quad \forall x \; P(x,x) , Q(a) \}$.

R(a) follows from the theory so we would want to be told that x=a is an answer to "R(x)?" .

We shall call any language which uses this declarative interpretation a *pure* logic programming language. - "pure" because it is purely declarative.

In [Kow74] and [CKRP73], it was shown how such programs could be used as computer instructions - i.e. given a procedural interpretation - by using automated theorem proving techniques based on the resolution principle introduced in [Rob65]. We shall look at the details of this approach shortly.

Because logic programming is very declarative, the relation between the problem

described by a logic program and the problem solved by running it is very clear cut. So it may seem unnecessary for us to look at the procedural details. Apart from completeness of exposition, there are two main reasons why we are doing so :

1) we want to distinguish PROLOG from pure logic programming;

2) in the next section we want to introduce a non-declarative extension (negation as failure) of logic programming and this can really only be introduced as a modification of the procedural interpretation of the language and ;

3) later in the thesis we want to introduce our own modification of the procedure to enable extra expressiveness.

So let us now look at the basis for a procedural interpretation of logic programs - what is called SLD-resolution (Linear resolution with a Selection function for Definite clauses). The interested reader is referred to [Kow79] or [Llo84] for alternative introductions.

The logic programming approach [Kow74] to positive query answering on Horn-Clause theories is based on the resolution principle of Robinson [Rob65]. We try to find a substitution α for the variables x in a query $\phi(x) = \wedge_{i=1}^{n} L_i$ such that the theory **P** is inconsistent with $\neg\phi(\alpha)$. If $\mathbf{P} \cup \{ \neg\phi(\alpha) \}$ is inconsistent then $\mathbf{P} \vdash \phi(\alpha)$ - so α is the type of answer we are looking for.

A *goal* from **LUC** is anything of the form $\leftarrow L_1, \cdots, L_p$ where p is a non-negative integer and each L_i is a literal with relation symbol from **L** and constants from **C**. When p=0 , the goal , "$\leftarrow$", is called the empty goal.

A resolution proof of the inconsistency of a Horn-clause theory P with the negation $\neg R(\alpha)$ of a substitution of a positive query $R(x)$ is basically a path of goals leading from the goal $\leftarrow R(x)$ representing the negation of the query to the empty goal. Each goal in the path represents a "less" inconsistent formula than a certain substitution of the last goal.

The empty goal represents the negation of truth i.e. falsity. A path from $\leftarrow R(x)$ to $\leftarrow$ with the substitution $x \rightarrow a$ shows that $\neg R(a)$ is "more" inconsistent with P than falsity i.e. $P \cup \{\neg R(a)\}$ is inconsistent. Thus $P \vdash R(a)$.

The attempted construction of a path of goals proceeds by the selection of a literal from the last goal and the choice of a clause of the program which says something about this literal (i.e. which has a head containing the same relation symbol). In SLD-resolution a rule governing the selection of literals is specified. The steps in paths of these versions must involve the literal indicated by the rule being used to get the next step. In most versions of PROLOG the selection rule is the "leftmost" selection rule.

We allow substitutions to operate on literals, queries and goals to get other literals, queries and goals in the obvious ways.

The crucial concept in the resolution process is that of unification. We say that literals L and K *unify* if there are substitutions θ , ϕ for their variables respectively such that $L\theta = K\phi$. θ and ϕ are called unifiers of L and K. For example, P(x,y,a) and P(u,u,v) unify with unifiers θ taking x to z and y to z and ϕ taking u to z and v to a.

We note that another but equivalent approach to unification is usually presented in literature on logic programming (for example in [Llo84]). If we assume that or make efforts to bring it about that L and K have no variable symbols in common then we can unify L and K by using just one unifier - $L\theta = K\theta$ say. We find that when we need to be precise in some later proofs the need to check on and change the symbols in the literals to be unified makes this alternate approach too messy. It is easy to see that our definition of unification is equivalent and we will see that it is less messy for our purposes.

If literals L and K unify then there are always *most general unifiers* (*mgus*) θ and ϕ - i.e. unifiers θ , ϕ such that , for all unifiers ψ, χ , for all variable symbols u,v, for all constant symbols a,

1) if $u\theta=v\theta$ then $u\psi=v\psi$

2) if $u\theta=a$ then $u\phi=a$.

3) if $u\phi= v\phi$ then $u\chi=v\chi$ and

4) if $u\phi=a$ then $u\chi=a$.

In the pure Horn-clause case unifications are useful because they help us use program clauses to get "less" inconsistent goals. Basically, if $\leftarrow L_1, \cdots, L_p$ is the most recent goal found in the construction of a path, and if L_i unifies with the head A of a clause $A \leftarrow A_1, \cdots, A_q$ with mgus θ , ϕ then $\leftarrow L_1\theta, \cdots, L_{i-1}\theta, A_1\phi, \cdots, A_q\phi, L_{i+1}\theta, \cdots, L_p\theta$ is "less" inconsistent with the Horn-clause theory than $\leftarrow L_1\theta, \cdots, L_p\theta$. For the exact formulation of this property see [Llo84] (or wait to section 5.1 where we give a generalised version).

In SLD-resolution with a selection rule **R** we end up with the following definition of a *successful* path for a goal G from a program PROG - otherwise known as an **R**-computed resolution refutation of PROG U {G}. A successful path for G from PROG with answer substitution θ is a finite sequence of goals $(G_0, G_1, \cdots, G_k)$ and substitutions $(\theta_0, \cdots, \theta_{k-1})$ such that

SLD1) $G=G_0$;

SLD2) G_k is the empty goal;

SLD3) for each i=0,1,...,k-1 , if G_i is $\leftarrow L_1, \cdots, L_p$ and **R** selects L_i then there is a program clause $A \leftarrow A_1, \cdots, A_q$ from PROG such that L_i and A unify with mgus θ_i, ϕ_i and G_{i+1} is $\leftarrow L_1\theta_i, \cdots, L_{i-1}\theta_i, A_1\phi_i, \cdots, A_q\phi_i, L_{i+1}\theta_i, \cdots, L_p\theta_i$; and

SLD4) θ is equal to the composition $\theta_0 \cdots \theta_{k-1}$ restricted to the variables in G.

If $\leftarrow L_1, \cdots, L_p$ has a successful path from PROG with substitution θ then we say that θ is an **R**-computed answer substitution for $L_1, \cdots, L_p$ from PROG.

It is not hard to show that for any selection rule **R**, this process gets exactly the

right answers according to the meaning we have determined for the program and query.

THEOREM 1.2 (from [Llo84] theorem 7.1)

For every pure logic program PROG, for every query $L_1, \cdots, L_p$, for every selection rule **R**, for every substitution θ of the variables in the query, if θ is an **R**-computed answer substitution for $L_1, \cdots, L_p$ from PROG then $\forall(\wedge_{i=1}^{p} L_i\theta)$ is a consequence of the Horn-clause theory represented by PROG.

THEOREM 1.3 (from [Llo84] theorem 9.5)

For every pure logic program PROG, for each query $L_1, \cdots, L_p$, for each selection rule **R**, for each substitution θ of the variables in the query, if $\forall(\wedge_{i=1}^{p} L_i\theta)$ is a consequence of the theory represented by PROG then there is an **R**-computed answer substitution σ of $L_1, \cdots, L_p$ from PROG and a substitution γ such that $\theta = \sigma\gamma$.

So now we have reduced the problem of giving a pure logic program a procedural interpretation to the problem of finding successful paths using any selection rule. As we have said, PROLOG uses the leftmost selection rule, but how does it find paths?

PROLOG proceeds by what is called a depth-first search with the choice of literals done on the basis of their order in the program. The computer tries to build one path at a time. Where it has a choice of clauses to unify a selected literal with, it chooses the first such clause in the program and extends the path using that clause. When it can extend the path no further, and yet it has not constructed a successful path, it goes back to try the other clauses at its most recent choice. This is known as backtracking. The full details of this standard control strategy are given in [Hog84].

One important thing to note is that PROLOG may miss out on finding some or all of

the successful paths.

What we have just described might better be called the procedural interpretation of "pure" PROLOG for nearly every existing version of PROLOG uses the syntax of pure logic programming as only a basis for a programming language. "Real" PROLOGs are described in [ClM81], [Hog84] and [CCP80]. In [Kow81b], Kowalski explains some of the main non-declarative (also called control or extra-logical) features of PROLOG. In the next section and throughout this thesis we will look closely at one of these features.

Finally, we shall answer the main question of this section - what is logic programming? Instead of giving it a rigorous definition, we use the term to refer to the general area of research that is wide enough to include both studies of purely declarative languages based on the language of logic and studies of non-declarative languages based on or related to PROLOG.

1.4 Negative Information

In the next few sections we will see how logic programming can be used in a database query language setting. One of the main themes of this thesis concerns the effect on expressibility of such query languages when their bases are extended from pure logic programming to logic programming based on negation as failure (NAF). What is NAF?

If we are going to use a logic program to describe a situation and a query then it is clear that we might be frustrated because we can not directly express rules involving negations. Even quite straight forward queries use negated conditionals in their definition. For example, you may want to describe a group of people who have parents, grandparents, sons and daughters amongst them and then add that a person's (blood) uncle is someone who is both a son of that person's grandparents and not that person's father.

NAF is an attempt to handle such negative conditions. It is not, as we will see, a very good attempt. Also, there are other ways of handling generalisations of Horn clauses. In particular, Hodges, in [Hod85], shows how to code any finite first-order theory as a Horn clause theory in such a way that we can use SLD-resolution on the latter to get answers to queries asked of the former. Nevertheless, the desire to work with negative information was the motivation for NAF.

Although algorithms which we would call versions of NAF were first appearing in automated theorem proving ([Hew72]) and logic programming ([Rou75] and [WPP77]) in the early and mid seventies, it was not until 1978 that serious attempts to justify the practice were made. The reason that problems of legitimation occurred was that NAF is a deviation from the principles of declarative programming - NAF fails as (an implementation of) negation.

In his detailed description of a *query evaluation process* (*qep*) - i.e. procedural interpretation - for logic programming with a special rule for handling negation, Clark in [Cla78] motivated the NAF rule by reference to the closed world assumption (CWA) which had appeared in [Rei78a] and [NiG78]. The CWA is the assumption that the person who wrote the program is implicitly giving a more detailed description than he explicitly gives : the facts which can not be deduced from his explicit description are in fact false.

It does not matter how we formalise this rough idea for we can now proceed to explain the motivation behind the NAF rule. A logic program will now be able to include clauses of the form $A \leftarrow A_1, \cdots, A_q$ where some of the A_i can be negative literals i.e. of the form not(P($\bar{x}$)) where P($\bar{x}$) is a positive literal.

When we come to do some serious work using NAF we shall describe a query evaluation process using a NAF rule (based on Clark's one) but for now we will just get a taste of its behaviour. Given a program and a goal we proceed as in SLD-resolution trying

to find a successful path from the goal to the empty goal. When a positive literal is selected then things proceed as before but when a negative literal is selected then the NAF rule goes into operation. For now we will assume that the literal is ground - say that it is not($P(\sigma)$). Now we leave this refutation for the moment and separately try and see whether $P(\sigma)$ succeeds or not by the same process. If $P(\sigma)$ succeeds then we go back to our main path and act as though not($P(\sigma)$) has failed - i.e. the path can not continue. If $P(\sigma)$ fails then, in accordance with the CWA intuition, we act as though not($P(\sigma)$) succeeded - i.e. the next goal in the path is the old one minus not($P(\sigma)$). In fact we don't really always know whether a query will not succeed because the process might get stuck in a loop. For this reason, we can only succeed not($P(\sigma)$) if $P(\sigma)$ fails in a finite amount of time and for this reason NAF should properly be called negation as finite failure.

For example, consider the following program :

$R(y) \leftarrow P(x,y)$, not($Q(x)$)

$P(a,b) \leftarrow$

$Q(b) \leftarrow$

From this program it is not hard to see that R(b) succeeds according to the NAF rule. This is because not(Q(a)) succeeds when Q(a) finitely fails as it does.

This example also shows that NAF is not an implementation of classical negation. What does it implement?

We have just managed to ask the peculiar question which many logic programmers find themselves asking. It is nothing special for computer scientists to be trying to develop some technology to solve a given problem. Here, however, we have the opposite. The technology has been developed but we are not sure about what the problem is that it solves. In fact, as we will see, no-one has yet succeeded in giving a satisfactory account of what NAF does implement.

There is no guarantee that such a declarative meaning for NAF actually exists but it would be useful if it did. Giving some meaning to NAF outside the complicated query evaluation process would be welcomed by the users of logic programming. Questions about whether a particular program truly implements a particular problem could be answered by comparing the problem with the meaning of the program and using general results about how well a program implements its meaning. An account of declarative meaning would also be useful to us when we want to generalise about the behaviour of logic programs.

Let us just be clear about what we would want from such a meaning. The meaning of any logic program will be a set of sentences of some language. We want it to be easy to construct from any program - the easier the construction is, the more declarative is the meaning. We will also give each query a formula as a meaning. In the pure logic programming case we have already specified what set of sentences a program represents and what sentence a query represents.

The important aspect is that the meaning must represent a problem which the program solves. The computer can give a substitution as the answer to a query, say "no" or get stuck in a loop (we will also see that some implementations of NAF can also give up). However, the meaning of a query can either follow from the meaning of the program or not. Obviously we will want a yes answer to correspond to a valid deduction and a no to correspond to not-deducible but we have to decide what a non-termination means.

Different people with different requirements may be happy with different accounts of declarative meaning depending on exactly how well they want a program to implement its meaning. We shall identify several types of relationships which could exist between a program an its meaning.

To make the ideas clear let us suppose that we are only asking ground queries with

ground meanings so we only get "yes" and "no" answers instead of substitutions. Also, we will only consider atomic queries and we will assume that the accounts of meaning are nice enough to define the meaning of such a query as the query itself.

It is common in logic programming research to see the term soundness used in this connection. We will use it for our weakest relationship between a program and a meaning. Let us say that a program *soundly* implements a theory if and only if
if a query succeeds from the program then the the query is deducible from the theory.

To know that a program soundly implements a theory can sometimes be useful - we know that a yes answer means yes. However, there are problems both with using it as a requirement on a practical particular level and as a requirement on the general level for an account of meaning.

First let us consider the practical level - solving a given problem. Clearly, soundness is a necessary requirement for a proposed solution to be acceptable: it would not usually be desirable for a program to come up with incorrect yes answers to the problem. On the other hand, in many circumstances, it is equally desirable for the program not to come up with incorrect no answers. The government might be happy for a computer, which soundly implements British entry-visa requirements, to say "no" to people who are entitled to have one, but the people might not be so happy. Requiring only soundness makes life easy for the programmer - she can implement any problem by a program which always says "no".

On the general level, a requirement of only soundness of an account of meaning is equally easy to satisfy. We could say that the meaning of any program is an inconsistent theory. Then, every program implements its meaning soundly. Clearly, any account of meaning which can only ensure that each program soundly implements its meaning is of little use to us.

Of much more use is the relationship which we will call correctness. Let us say that a program *correctly* implements a theory if and only if

1) if a query succeeds from the program then the query is deducible from the theory and

2) if a query finitely fails from a program then the query is not deducible from the theory.

So here we are interpreting a loop as meaning don't know - we do not know whether the meaning of the query is deducible from the meaning of the program or not.

On the general level, correctness is the ideal relationship. Given a program, if our general account of meaning provides it with a meaning which it implements correctly, then no-one is to say that the program is not an attempt to implement that theory. Nothing in the behaviour of the program contradicts this assumption. A general account which provides every program with a meaning which it implements correctly is very satisfactory.

For the purposes of solving problems, on the other hand, correctness is not sufficient. The programmer can easily provide a program which correctly implements any problem: he just has to use the program which loops on all the queries.

We need a stronger notion still. Let us say that a program completely implements a meaning if and only if

a query succeeds from the program if and only if the meaning of the query is deducible from the meaning of the program.

It should be clear that completeness implies correctness.

Correctness is all that we require for our reverse problem of finding a meaning for given programs. For the usual problem of solving problems, we do not need an account of meaning which always gives a theory which the program implements completely. It is sufficient that we show that under certain conditions on the program, the program

implements its meaning completely. The programmer may then be able to solve a problem by presenting a program which has a meaning which represents the problem and which satisfies the conditions. Then our general result implies that the program gets all the yes answers and no more.

We could go a step further and define perfection to be completeness plus no looping - i.e. the answer to all queries is either a correct yes or a correct no. However, logic programming with NAF rarely obtains the necessary conditions of no looping for this relationship to hold - whether or not we have an account of meaning.

We have said that there are no accounts of the meaning of logic programs with NAF. Yet, at least two ideas are commonly put forward - the CWA and Clark's Completed Database (CDB). Let us see why they are unsatisfactory.

The following material is based on the work of Shepherdson in [She84] and [She85] and Flannagan [Fla86] in which papers the relationship between Clark's qep, the CWA and the CDB is explored in detail.

A simple example shows that the CWA does not give us the meaning of NAF. Consider the following program.

$P \leftarrow not(Q)$

Neither P nor Q follows from the program (taken as the set $\{ \neg Q \rightarrow P \}$ of sentences) so the implicit meaning of the program under the CWA contains $\{ \neg Q \rightarrow P , \neg P , \neg Q \}$ from which Q is a consequence (as the set is inconsistent). However Q finitely fails from the program so the program incorrectly implements its supposed meaning.

Contrasted with that observation that the CWA does not give a basis for finding the meanings of programs is the following result.

THEOREM 1.4 (theorem 1 [She84])

If a query Q fails in the qep then its negation $\neg Q$ follows from the program plus the CWA and if Q succeeds with answer θ then $Q\theta$ follows.

This might seem to supply us with something as good as a correctness result. However, the relationship which theorem 1.4 shows holds between a program and its CWA meaning suffers the same defects as soundness. First, any always looping program implements any problem. Second, giving any program an inconsistent meaning is just as good, and a lot simpler, than giving it the CWA meaning. Theorem 1.4 is not good enough to justify the use of the CWA as a basis for an account of declarative meaning of NAF.

The other main contender for the title of the meaning of NAF is the CDB idea which Clark introduced with his qep in [Cla78]. Although the failings of the CDB are much the same as those of the CWA it is worth describing it in greater detail because something like it will make an appearance in this thesis.

Constructing the CDB is essentially replacing the "if"s in the program by "iff"s. More precisely we proceed as follows.

First of all, we decide on a language **L** which contains not only those symbols in the program but those which we might want to query about. As pointed out in [Fla86] , Clark makes his "construction" of the CDB impossible by not limiting his CDB to a particular language. The CDB actually depends on this choice as well as on the program.

For each relation symbol R from **L**, we put one sentence, called the definition of R, in the CDB. If R is n-ary we choose n variable symbols $\boldsymbol{x} = (x_1, \cdots, x_n)$, not appearing in the program. If R appears in the heads of no program clauses then the definition of R is

$$\forall \boldsymbol{x} \; (\; \neg R(\boldsymbol{x}) \;).$$

Otherwise, if R appears in the heads of precisely clauses c_1 , $\cdots$, c_k in the program for k > 0 then, for each such clause c_i , R(t) $\leftarrow L_1, \cdots, L_p$, say, we first construct a formula $E_i = E_i(x)$. If the distinct variable symbols in the clause are y then

$E_i = \exists y(t_1 = x_1) \wedge \cdots \wedge (t_n = x_n) \wedge L_1 \wedge \cdots L_p$.

Now we define the definition of R to be

$$\forall x[\mathrm{R}(x) \longleftrightarrow E_1 \vee \cdots \vee E_k].$$

Although, unlike Clark, we are using the first-order theory with identity so we do not have to add equality theory to get the same answers as Clark, we do have to follow him in specifying the free interpretations for terms. So, to the definitions above we add

$a \neq b$ for each pair of distinct constants from **L**,

$\forall xy(f(x) \neq g(y))$ for each pair of distinct function symbols f,g from **L**,

$\forall xy(f(x) = f(y) \rightarrow (x_1 = y_1 \wedge \cdots \wedge x_n = y_n))$ for each function symbol f from **L**,

$\forall x(f(x) \neq a)$ for each constant symbol a and function symbol f from **L**

and finally we add $\forall x(t \neq x)$ for each term t built from at least one function symbol and each x which is one of the variables x in t.

In [Cla78] is proved the same sort of result for the CDB as Shephedson's result for the CWA. Again the observation that meaning as contradiction would do just as well makes the practical use of this doubtful.

An example shows that there is no hope for a general correctness result for the CDB either.

P $\leftarrow$ not(P)

is a program which finitely fails the query "S?" and yet S follows from the inconsistent CDB of this program.

To sum up then, NAF can only be seen as a non-declarative feature of a programming language. To find out what problem a given logic program with NAF is

implementing we will have to follow through how the computer would run the program. Contrary to what we have seen in the pure logic programming case, currently there is no satisfactory way of avoiding this by referring to a separate meaning of the program.

1.5 Deductive databases

In the next two sections we will look at how logic programming can be applied to our problem of answering queries on finite relational structures. One general framework for doing this is the deductive database approach represented well in a series of collections [GaM78] , [GMN81] and [GMN84b]. In this section we will see what deductive databases are in theory and in practice.

In its most general form (see for example [Kow81a]) a deductive database is the representation of a set of sentences (or sometimes the theory of consequences of a set of sentences). Having the conventional / deductive database dichotomy paralleling the model-theoretic / proof-theoretic dichotomy of logic like this has many consequences for the way we deal with and talk about the two types of database. [NiG78], [Kow81a] and [Rei84] contain discussions of these consequences.

Thus a deductive database is a not necessarily complete description of a situation and there may be several essentially different situations which it describes. Queries no longer get just yes or no answers. In a conventional database representing a structure D we either have $D \models \phi(\bar{a})$ or $D \models \neg\phi(\bar{a})$. However if we have a deductive database representing a theory T then there are four different possible relationships:

1) $T \vdash \phi(\bar{a})$ and not $T \vdash \neg\phi(\bar{a})$ - the yes answer;

2) $T \vdash \neg\phi(\bar{a})$ and not $T \vdash \phi(\bar{a})$ - the no answer ;

3) neither $T \vdash \phi(\bar{a})$ nor $T \vdash \neg\phi(\bar{a})$- the don't know answer; and

4) both $T \vdash \phi(\bar{a})$ and $T \vdash \neg\phi(\bar{a})$ - the inconsistent answer.

Being able to deal with situations of which we do not have complete information is one of the advantages of the deductive approach over the conventional approach. The conventional approach has been modified to deal with incomplete information by the use of some three valued logic (see [Cod79] and [Lip79]) but this approach is not very satisfactory ([Gra77] , [Vas79] , [Vas80]). The other big advantage is that we can describe information using general rules instead of lists of facts - and thus save on space and make the database nicer for humans to deal with.

That's the theory of deductive databases. Now let us look at the practice. There are at least three ways in which the research and use of deductive databases differ from the full generality of the ideas we have just seen.

First, not just any set of sentences is acceptable as a deductive database. Obviously, if we want to find out what follows from the sentences then we want them (and the query) to be in an acceptable form for automated theorem proving. The first papers introducing the deductive database idea (for example [Emd78] and those in [GaM78]) did not distinguish it much from logic programming. As Harel noted in [Har80], the deductive database researchers were a relatively self-contained group who had more to do with logic programming than database research. Given this history it is not surprising that we often see deductive databases being introduced in the form of logic programs ([GMN84a] , [LlT85] and [LlT86]). The actual job of using PROLOG to answer deductive database queries is described in [WaP81] , [Dah82] , [Llo83] , [LlT85] and [LlT86].

Second, not just any formula is usually acceptable as a query to a deductive database. Again, we make the restriction to what logic programming can handle - conjunctions of literals. This is not such a serious limitation as it might at first seem because the ability of deductive databases to contain rules means that queries can be defined inside the database. To be more precise, suppose that we know that we are going to want to query the database about the formula ϕ. Then , provided we can express ϕ in

the restricted form appropriate to the database, we can choose a new symbol P say and describe P, inside the database, as having interpretation those tuples which satisfy ϕ. Then we only need to ask "P(x) ?".

Third is the approach to negative information. It would be quite restrictive to stick to descriptions of situations and definitions of queries which are expressed in pure logic programming. All the answers to the questions we can ask will either be the yes answer or the don't know answer. A negated atomic sentence $\neg P(x)$ is never the consequence of a set of Horn clauses. However, as we saw in the last section, we must be careful when we step outside this restricted domain.

In practice the attempted solution to this problem is usually some use of NAF (see for example [Kow78]). Now there is some cause to combine use of the CWA with pure logic programming - as we will see in the next section - but it is not satisfactory to use NAF for this purpose even in this restricted setting and there certainly are problems with a more widespread use of the rule.

Some of the properties of NAF in the pure logic programming case are described in [ApE82] and [JLL83] but they do not equate the CWA with NAF. Gallaire, Minker and Nicolas in [GMN84a] try to justify the use of NAF in the pure logic programming case - i.e. a situation is described in some way by a pure logic program and then when queries are asked and fail the answer is taken to be "no". Unfortunately, it is mistakenly assumed that the CWA , the CDB and NAF are all effectively the same thing in this case. So their formalisation is incoherent. In particular, their belief that the CDB is complete (in the model-theoretic sense) leads them to make several false claims.

Consider the following deductive database:

edge(a,b)

edge(b,b)

TC(x,y) ← edge(x,y)

TC(x,y) ← edge(x,z) , TC(z,y)

By applying Clark's completion and other rules (as in [GMN84a]) we get the following theory **P** as being the theory which the database represents:

$\forall x\ (x=a) \vee (x=b)$,

$a \neq b$,

$\forall xy\ [\ edge(x,y) \longleftrightarrow ((x=a \wedge x=b) \vee (x=b \wedge y=b))]$,

$\forall xyz\ [\ TC(x,y) \longleftrightarrow (\ edge(x,y) \vee \exists z\ (\ edge(x,z) \wedge TC(z,y))\ )\ ]$.

In [GMN84a] it is claimed that a theory derived in such a way from a deductive database will always be complete. Unfortunately neither TC(b,a) nor ¬TC(b,a) are derivable from **P** - **P** is not complete. So even if NAF could be used to find all those (x,y) such that ¬TC(x,y) is deducible from **P** then it still would not implement the CWA.

It must also be pointed out that relying on the CWA for any justification deprives deductive databases of their ability to describe situations incompletely. Of course, that is fair enough if , like us, one is concentrating on completely known structures.

Another justifying-the-technology approach can be found in [LlT85]. Lloyd and Topor consider databases consisting of clauses of the form A ← W where A is an atom and W is a typed first-order formula. The stance is very much one of finding a declarative meaning for a procedural interpretation based on conversion (following results in [LlT84]) of the database into a logic program and then the usual NAF qep. Such a "here's the solution / what's the problem ?" approach can be useful if the user is told when she or he can and how he or she can rewrite a given problem in the form of the declarative meaning of a program, construct such a program and run it to get the right answers. Lloyd and Topor, who are essentially using a CDB justification for NAF, do not give instructions for converting a given problem into the CDB of a program and leave the question of when the program gets the right answers as an open problem.

It is shown (in theorem 1 [LlT85]) that each "yes" answer which the qep gets is a correct answer according to the meaning. However, as we have remarked before, this is nothing special unless we know that a "no" answer means "no".

Now let us skip over this swampy area of justifications for NAF and assume that either we are restricting ourselves to pure logic programming, that someone has come up with a declarative meaning for NAF or that someone has extended pure logic programming in some other declaratively justified way. Let us see how we apply the deductive database approach to our problem of asking queries of completely known finite relational structures.

We have seen that queries are often defined as derived relations inside deductive databases. When a deductive database is used to ask a perhaps complicated query to a finite structure then this leads to the existence of two different sets of relation symbols in the database : there are the symbols from the signature of the structure which we will call *extensional* symbols and there are the new symbols which we have introduced to help us define the query and which we will call *intensional* symbols.

In general, the clauses defining the extensional symbols will be facts i.e. ground clauses with empty bodies, while the intensional symbols will be in the heads of rules i.e. not facts. Reiter in [Rei78a], [Rei78b], [Rei78c] and [Rei80] has done a lot of work based on the idea of a partition of the database into an extensional database (EDB) consisting of facts and an intensional database (IDB) consisting of any types of clauses. Minker has shown, in [Min82], that it is easy to equivalently rewrite any deductive database in this form.

In our applications the EDB will contain a description of the queried structure. If L is a signature and D a finite L-structure then the EDB describing D consists of all the clauses $R(\bar{a}) \leftarrow$ such that R is n-ary from L, $\bar{a}$ is an n-tuple from the domain of D and

$D \not\models R(\sigma)$. We assume that D is nice enough to have a domain which consists of symbols which we can use in a logic program.

Maybe we can represent our query in the IDB in some way but we are still faced with the problem that in the pure logic programming case we are stuck with getting only "yes" or "don't know" answers. In the next section we will see how we can effectively use the CWA to get us some "no" answers.

1.6 Datalog

From beginnings within deductive database theory another approach to the application of logic programming to database problems has emerged. This is the use of the Datalog, or Horn clause, query language which forms a basis for the research in this thesis.

Datalog can be thought of as the EDB/IDB approach in the pure logic programming case in combination with the CWA. So, although the CWA will be used, all the logic programs in this section will be pure - i.e. no NAF. We shall now make some other restrictions on the IDB which we will explain in the next chapter : we suppose that no constant or function symbols appear in the IDB, and as usual, no extensional relation symbols appear in IDB heads. Not all of these restrictions are always made in every paper on Datalog but they all are in [ChH85] for example.

A Datalog query $(P;P)$ will have its answers defined as the interpretation of a particular symbol P by such an IDB, P, along with what is effectively a use of the CWA.

DEFINITION

If **L** is a signature then DATALOG(**L**) is the **L**-query language specified below.

SYNTAX:

(P;P) is a wff of DATALOG(L) if and only if

there is a signature S disjoint from L such that

1) P is a finite set of L∪S Horn clauses with only symbols from S appearing in their heads and

2) P is a relation symbol from S.

SEMANTICS:

If P and P are as above with P n-ary then the wff (P;P) represents the n-ary L-query q where

for all finite L-structures D,

if EDB = { $R(\bar{a}) \mid D \models R(\bar{a})$, $\bar{a}$ is from D , R from L },

then for all $\bar{a}$ from D^n,

$\bar{a} \in q(D)$ if and only if EDB $\cup P \vdash P(\bar{a})$.

One way of getting answers to a DATALOG(L) query (P;P) asked of a finite L-structure D is as follows.

1) describe D by EDB,

2) combine EDB with the IDB, P, to get a pure deductive database,

3) query the database with "$P(\bar{x})$?",

4) and if $P(\bar{a})$ succeeds then $\bar{a}$ is a correct answer to the query and otherwise, and here it looks like we are using the CWA, $\bar{a}$ is not a correct answer to the query.

Let us see Datalog in action. As commented on in [GMSV87], the "canonical" example of a Datalog query is (P;path) which computes the directed transitive closure of edge on any finite {edge}-structure when P is as follows :

{ $\forall x$ (edge(x,y) $\rightarrow$ path(x,y)),

$\forall xyz$ (path(x,z) $\wedge$ path(z,y) $\rightarrow$ path(x,y)) }.

For instance, if D contains a, b, c and d and the only edges are from a to b and from b to c then to find whether (a,c) is in the transitive closure of edge in D we will use "path(a,c) ?" to query the following database :

edge(a,b) ←

edge(b,c) ←

path(x,y) ← edge(x,y)

path(x,y) ← path(x,z) , path(z,y) .

Clearly the querying succeeds as it should. Since "path(d,a) ?" does not succeed we can conclude that (d,a) is not in the transitive closure. The fact that path(d,a) does not finitely fail shows that we are using the CWA and not NAF.

Datalog has been presented here as a specialised part of deductive database research. However, there are at least three reasons for regarding it as distinct in practice.

First, there is no necessity for the logic programming procedural semantics to be used for Datalog query evaluation and, in practice, it is not used much (although [JCV84] is an exception). More common (see for example [HeN84], [Cha79] , [McKS80] , [McKS81]) is the compiling approach where the description of the queried structure is stored in a conventional database and the query, (P;P), is compiled - i.e. transformed - into a program of conventional database access instructions which can then be applied to the database. Also based on storing the structure in a conventional database is the bottom-up approach (see for example [Ull85]) where the derived relations defined by a set of sentences P are built recursively as whole relations (as opposed to the top-down one tuple at a time approach of logic programming).

Second, Datalog is a query language so there is much more emphasis on questions of optimising queries and expressibility (see for example [BaR86]) than there is in deductive database research where queries are hidden as derived relations inside databases. [GMSV87] gives a short summary of such Datalog work before going on to investigate the

practicality (or otherwise) of one approach to the problem of optimising Datalog queries. [ChH85] considered the expressibility of Horn clause queries and in particular compared them with fixed point languages. It was shown that Datalog expresses precisely those queries representable by a fix point operator applied to a positive existential query. We shall look at their work in more detail in the next section.

Third, the semantics of Datalog based on what is provable distinguishes it from the semantics of deductive databases based on what is true. In its theoretical form a deductive database is a theory which may describe a situation. If the theory is consistent then we can deduce that some facts are true in a model of the theory, we can deduce that some facts are false and we do not know about some other facts perhaps. From a theory, on the other hand, each fact is either provable or not provable. Datalog is based on this latter relationship.

What has just been said does not seem to agree with the introduction of Datalog as pure deductive databases with IDB/EDB divisions and the use of the CWA. It might also be noted that the CWA does not appear much in Datalog literature, and indeed it does not appear in our definition of Datalog. These observations can be explained by the fact that the provability semantics of Datalog and the pure logic programming plus CWA semantics are two different means to the same end. Each fact $P(\bar{a})$ is either provable or not provable from a given theory $P \cup EDB$. If it is provable then $\bar{a}$ is a correct answer to the Datalog query (P;P) asked of the structure D described by EDB. But also $P(\bar{a})$ will be true in any model of $P \cup EDB$, so the logic programming semantics says "yes" too. If $P(\bar{a})$ is not provable from $P \cup EDB$ then $\bar{a}$ is not a correct answer to (P;P) asked of D. But also, since $P(\bar{a})$ is not provable, the CWA says that we assume that $P(\bar{a})$ is false in a model implicitly described by $P \cup EDB$ under the CWA - so the logic programming semantics says "no" too.

In this thesis we will try to do all of our defining of queries explicitly and not leave work to be done by implicit assumptions like the CWA.

1.7 Extending Datalog

We have seen how we can have a query language based on logic programming. In this section we will argue that both the motivation and the possibility exist for defining a practical query language which is also based on logic programming but is more expressive than Datalog.

We have said that Chandra and Harel have shown in [ChH85] that Datalog expresses precisely the queries representable by a fixed point operator applied to a positive existential first-order formula ([ChH85] Theorem 4.1). Thus DATALOG(L) is a (not necessarily strictly) less expressive language than FO+LFP (L).

In fact, Chandra and Harel point out two very simple and useful classes of FO+LFP queries which are not expressible in Datalog. These are the complement and the universal quantification of an extensional relation - e.g. asking " $\neg$ R(x)?" and "$\forall$ y R(y)?" of {R}-structures. We will show that these are not in Datalog later (but it is easy to do). Using the same ideas it can be shown that asking "whose brother is liked by everyone?" of a {parent, male, likes }-structure can not be done in Datalog. Other queries involving universal quantification, negation and inequality appear in all sorts of practical applications.

There is, thus, ample justification for regarding Datalog as inadequate to some extent in practice and certainly inadequate compared to FO+LFP.

Chandra and Harel in [ChH85] went on to define several extensions of Datalog. We shall meet their extensions H', C and FN later in the thesis. Apart from brief references to

the possibility of implementing C using NAF and the apparently severe problems implementing FN, Chandra and Harel concentrate on the expressive power of their extensions. We, on the other hand, being interested in query languages based on the automated theorem-proving logic programming techniques will pay much more attention to implementational aspects.

The natural question for us to ask at this stage is "can the computational rule NAF help us answer more questions? ". It should be possible to modify the logic programming approach to answering Datalog queries to allow impure programs. We have seen that Datalog queries can be implemented by sticking a pure logic program after a description of the queries structure. We shall refer to the members of a larger class of queries which can be implemented in the same way by sticking an impure logic program after a description of the queries structure as LPNAF queries (but we will leave it until later to define this class rigorously).

So it is possible to extend Datalog but the problems we have seen with giving declarative meaning to NAF all get in the way of working out what queries we can now answer.

For example, we can see that we can answer the popular brother query. Suppose that we have a finite { parent, male, likes }-structure D. We first describe D in a logic program as follows.

1) We introduce a new 1-ary symbol exist.

2) If $\{a_1, \cdots, a_k\}$ is the domain of D (in some order) we put the clause

" exist(a_i) ← " in the program for each i.

3) If D ⊨ parent(a,b) (meaning a is a parent of b) we put " parent(a,b) ← " in the program.

4) If D ⊨ male(a) (meaning a is a male) we put " male(a) ← " in.

5) If D ⊨ likes(a,b) (meaning a likes b) we put " likes(a,b) ← " in.

To such a description of any structure we add the following clauses using new symbols 2-ary eq and neq and 1-ary unpop and popbroth.

eq(X,X) ←

neq(X,Y) ← exist(X) , exist(Y) , not(eq(X,Y))

unpop(X) ← exist(X) , exist(Y) , not(likes(Y,X))

popbroth(X) ← parent(Y,X) , parent(Y,Z) ,
male(Z) , neq(X,Z) , exist(Z) , not(unpop(Z))

It should be apparent that under such a set up, popbroth(c) succeeds under the NAF rule if and only if c has a brother who is liked by everyone.

In [ChH85] Chandra and Harel say that NAF justifies their extension of **H** to **C** from an operational point of view and that NAF gives rise to the possibility of using the language **C**. Later we will try to formalise exactly what they are claiming here and show that it is unjustified. Although it is easy to devise a method for computing answers to C queries, this is not using anything like the procedural interpretation (i.e. query evaluation process) of logic programming and we will show that it can not be maintained that C gives declarative meaning to LPNAF theories.

We are left with some open questions.

1) Is there a natural extension of Datalog which captures the expressive power of LPNAF - (preferably in as declarative a way as possible)?

2) If so, then how does the expressive power of this language compare with other languages - in particular FO+LFP which contains lots of useful queries?

Let us get to work on these.

Chapter 2

Provability

In this chapter, we shall establish a general framework for query languages based on the provability semantics. We shall first define a query language, PROV, formalising the full generality of this idea. Then, after showing that PROV is very expressible, indeed too expressive to be practical, we will consider restrictions to various sublanguages. By being more restrictive we can limit ourselves to easier queries to compute. The least expressive sublanguage we consider is Datalog itself. Then we gradually add more features to increase expressibility until we reach PROV again. Questions of expressibility and implementability of one sublanguage, DEF, will be left for later chapters.

2.1 PROV

In this section we will try to capture the full generality of the provability semantics in a very expressive query language. This is so that we can later switch our attention to less expressive, more easily implementable languages just by defining restricted sublanguages of the full language.

In describing the language Datalog we have already given some meaning to the term "provability". In particular we contrasted it with definability. Let us have a quick look at definability again.

The concept often appears in logic with the presentation of the Beth definability theorem - see for example [ChK73]. It concerns the relationship between two types of

definability - explicit and implicit. We will need to use versions of these concepts specially adapted for the query language setting.

Definability in this context concerns how precisely a particular theory determines the relationship between interpretations of its symbols in its models. For our purposes we consider a theory with two classes of relation symbols - the old and the new. Suppose that **L** and **S** are two disjoint signatures and that **P** is an **L**$\cup$**S**-theory. Do **P** and interpretations for the old symbols - those from **L** - determine an interpretation for one of the new symbols P from **S**?

We will say that **P** explicitly defines P if there is an **L**-formula ϕ such that
$\mathbf{P} \vdash \forall \boldsymbol{x}\ (\ P(\boldsymbol{x}) \longleftrightarrow \phi(\boldsymbol{x})\ )$.
In this case we know exactly and explicitly what the interpretation for P is in terms of the interpretation for the symbols from **L** in all models of **P**.

However, maybe **P** specifies exactly the interpretation for P for fixed interpretations for the symbols from **L** but not necessarily in an explicit way. Maybe if E and F are two models of **P** which agree on interpretations for the symbols from **L** - i.e. $E|_L = F|_L$ - then E and F also agree on the interpretation for P. We can say this equivalently in a syntactic way by saying that if the **L**$\cup$**S'**-theory **P'** is got from **P** by replacing each symbol S from **S** by a symbol S ' of the same arity from **S'** then
$\mathbf{P} \cup \mathbf{P'} \vdash \forall \boldsymbol{x}(\ P(\boldsymbol{x}) \longleftrightarrow P'(\boldsymbol{x})\ )$.
In this case we will say that **P** implicitly defines P.

The Beth definability theorem concerns the relationship between these two notions of definability.

THEOREM 2.1 (from [BoJ74])

If **L** and **S** are two disjoint signatures, **P** is an **LUS**-theory and P is from **S** then **P** explicitly defines P (in terms of the symbols from **L**) if and only if **P** implicitly defines P (in terms of the symbols from **L**).

It is easy to see how to use ideas from here to employ theories to define queries. If **L** and **S** are disjoint signatures and **P** is an **LUS**-theory which implicitly (or equivalently explicitly) defines P from **S** then, given any finite **L**-structure D, there is at most one way of interpreting P in expanding D to a model of **P** - otherwise we could construct two models of **P** which agree on their **L**-reducts (both D) but disagree on P. So, provided we could guarantee that there will be at least one expansion of each such D to a model of **P**, then we have the possibility of a query language - the answer to the definability query determined by **P** and P, when asked of D, is the relation which P must be interpreted as in any expansion of D to a model of **P**. In other words **P** and P determine a query q such that $a \in q(D)$ if and only if, for every expansion of D to a model E of **P**, $E \models P(a)$. We can replace "every" here by "some" without changing q because of the conditions on **P** and P.

If **P** is

$\{\forall x\,(R(x) \vee S(x))\,,\ \forall x\,(T(x) \longleftrightarrow ((R(x) \wedge P(x)) \vee (S(x) \wedge P(x))))\}$

then **P** explicitly defines T in terms of {P} (as P). It is also true that any finite {P}-structure D can be expanded to a model of **P** - simply make R true on all elements, S not true anywhere and T have the same interpretation as P. Not surprisingly, **P** and T determine a 1-ary query with correct answers all those c such that $D \models P(c)$.

The provability idea is just an extension of this semantics to the case where we drop the restrictions on **P** and P. No longer do we care whether **P** defines P, we just see what

elements are in every expansion of D to a model of **P**. If **P** has no such expansion then all the tuples are.

Suppose that we have a signature **L**. Let us define PROV(**L**).

SYNTAX of PROV(**L**)

If **S** is a signature, n a non negative integer, **P** a finite **L**∪**S**-theory (written in some order)

and S an n-ary symbol from **S** then PROV(**P**;S) is a wff of PROV(**L**) and such wffs are the only wffs.

SEMANTICS of PROV(**L**)

If PROV(**P**;S) is the wff above then for all finite **L**-structures D, for all $\bar{a}$ from D^n, $D \models \text{PROV}(\mathbf{P};S)(\bar{a})$ if and only if for every expansion E of D to a model of **P**, $E \models S(\bar{a})$.

For example, if **L** contains 2-ary P, **S** contains 2-ary T and **P** is the **L**∪**S**-theory

$\{ \forall xy (P(x,y) \rightarrow T(x,y)) , \forall xyz (P(x,z) \wedge T(z,y) \rightarrow T(x,y)) \}$

then PROV(**P**;T) is a 2-ary **L**-query which has correct answers on a finite **L**-structure D, those ordered pairs in the (directed) transitive closure of the interpretation of P in the structure.

It might seem unnecessary for us to have restricted **P** to a relational theory. However, for reasons of implementation of sublanguages which we will mention later it is best if constants and function symbols do not appear in **P**. This does not cost us anything in expressivity because it is easy to do the usual trick of rewriting theories in an equivalent form with only relation symbols (although we need some new relation symbols).

2.2 Provability and Minimal Models

For an example let us look at the following theory. **L** has 0-ary raining, **S** has 0-ary roof-leaking, 1-ary outside, inside and wet,

T={ raining→roof-leaking,

$\forall x$ [raining $\wedge$ outside(x) → wet(x)],

$\forall x$ [roof-leaking $\wedge$ inside(x) → wet(x)],

$\forall x$ [outside(x) $\vee$ inside(x)] }.

Now suppose we have an **L**-structure D having domain with one element Richard and the 0-ary relation raining being true in D. Then D ⊨ PROV(**T** ; *roof-leaking*) and D ⊨ PROV(**T** ; *wet*)(Richard) but we don't have D ⊨ PROV(**T** ; *outside*)(Richard) nor do we have D ⊨ PROV(**T**; *inside*)(Richard).

We can use the notation to define new relations on the given structure. In that example we have virtually defined a new 0-ary relation "ROOF-LEAKING" and new 1-ary relations "WET", "INSIDE" and "OUTSIDE" on D. ROOF-LEAKING is the 0-ary relation true , WET contains Richard and INSIDE and OUTSIDE are both the empty 1-ary relation. Given any **L**-structure D' we can define corresponding relations ROOF-LEAKING', WET', OUTSIDE' and INSIDE' by

ROOF-LEAKING' ⟺ D' ⊨ PROV(**T**; *roof-leaking*),

and for any d ∈ D' , d∈WET' ⟺ D' ⊨ PROV(**T**; *wet*) (d),

d ∈ INSIDE' ⟺ D' ⊨ PROV(**T**; *inside*) (d),

d ∈ OUTSIDE' ⟺ D' ⊨ PROV(**T**; *outside*) (d).

In general , if **L** and **S** are disjoint signatures, D an **L**-structure , **P** an **L**∪**S**-theory, and S an n-ary symbol from **S** then we can define an n-ary relation σ on D as follows:

$\forall \bar{a} \in D^n$, $\bar{a} \in \sigma$ ⟺ D ⊨ PROV(**P**;S) ($\bar{a}$).

Associating each n-ary relation symbol in **S** with an n-ary relation on D seems like

expanding D to an LUS-structure - one can interpret the symbol from S as its associate relation. The above example shows that this is not such a good idea. If we do expand D to an LUS-structure E along those lines then we find that $E \models \neg outside(Richard) \wedge \neg inside(Richard)$. Thus E is not a model of T. It is not very pleasing to use T to expand D and not end up with a model of T. This does not invalidate anything we have done but it is an example worth keeping in mind.

If an LUS-theory P is such that we can expand each finite L-structure D in this way and end up with a model E of P then clearly this is a minimal expansion of D to a model of P with respect to the interpretations of the symbols from S. Under such circumstances we will call P *minimisable*. Minimisable provability queries are much easier to understand. This is perhaps because its nicer to think about what is true in a model (in this case the minimal model) than what is provable from a theory.

2.3 Provability and Theorem-Proving

What has provability got to do with automated theorem-proving?

Let us suppose that we have an automated theorem prover i.e. a machine which, for some class T of theories, behaves as follows:
if U is a signature, C is a set of constant symbols, U is an n-ary relation from U, $\bar{c}$ is an n-tuple of constant symbols from C and Q is a first-order UUC-theory from T (written in some order) then the machine says "yes" to "Q ? U($\bar{c}$)" if and only if $Q \vdash U(\bar{c})$ and it says "no" otherwise. If it could exist then an implementation of PROLOG with a loop checker would work as such a machine for T being Horn-Clause theories.

Can we use our machine to find out whether $D \models$ PROV(P; S) $(\bar{a})$ or not? By definition $D \models PROV(P;S)(\bar{a})$ if and only if for every LUS-structure F , if $F|_L = D$ and $F \models P$ then $F \models S(\bar{a})$. If we could find some first-order way of saying that $F|_L = D$ then we could use

our machine. For suppose that F⊨**D** if and only if $F|_L$=D then

D⊨PROV(**P**;$S(\sigma)$)

<=> $\forall$**L**U**S**-*structures* F, F⊨**D**U**P** => F⊨$S(\sigma)$

<=> **D**U**P**⊢$S(\sigma)$

<=> the machine says "yes" to "**D**U**P** ? $S(\sigma)$".

Unfortunately we can not find any such **D** because a first-order theory can not define a domain except up to isomorphism. However that is all we need.

DEFINITION If **L** is a signature, D is a finite **L**-structure, and **C** is the domain of D regarded as a set of constant symbols then the facts about D, FA(D), is the **L**U**C**-theory DCA(D) U UNA(D) U PFACT(D) U NFACT(D) where

DCA(D) is $\{ \forall x \bigvee_{c \in C}(x=c) \}$,

UNA(D) is $\{ \bigwedge_{c \neq d \in C}(c \neq d) \}$,

PFACT(D) is $\{ R(\sigma) \mid D \models R(\sigma) \}$,

NFACT(D) is $\{ \neg R(\sigma) \mid D \models \neg R(\sigma) \}$.

It is straight forward to prove that if a finite **L**U**S**U**C**-structure G is a model of the facts about D then $G|_L$ is isomorphic to D as an **L**-structure. This is as close as we can get in first-order logic to forcing $G|_L$ to be equal to D.

A little model theory shows that

$\forall$**L**U**S**-structures F, if $F|_L$ =D and F⊨**P** then F⊨$S(\sigma)$

<=>

$\forall$**L**U**S**U**C**-structures G , if G⊨FA(D)U**P** then G⊨$S(\sigma)$.

This last result is equivalent to the following.

RESULT 2.2

$D \models PROV(\mathbf{P};S)(\sigma)$ is true if and only if $FA(D) \cup \mathbf{P} \vdash S(\sigma)$.

Thus our machine says "yes" to "$FA(D) \cup \mathbf{P} \; ? \; S(\sigma)$" if and only if $D \models PROV(\mathbf{P};S)(\sigma)$.

In order to compute with our new provability relationship a user has only to either use a slightly adapted theorem-prover or use an ordinary theorem prover but type in the facts about D followed by **P** when he or she wants to determine whether $D \models PROV(\mathbf{P};S)(\sigma)$ or not.

2.4 PROV and USO

It is easy to prove that PROV and the language of universally quantified second order queries USO are equally expressible. Chandra and Harel in [ChH85] defined a language FN very similar to PROV and showed that it is as expressive as USO. Their result is not very hard to prove or surprising either - all three languages are clearly equivalent. We justify defining and using PROV in this thesis rather than FN or USO because most of the other less powerful languages which we consider are sub-languages of PROV and not sub-languages of FN nor USO i.e. the less powerful languages use wffs from PROV representing the same queries as they do in PROV.

USO (and FN) are based on sentences. PROV is based on theories . Showing the equivalence of USO and PROV is not really much more than conjoining the sentences in a theory to form a long sentence.

DEFINITION of USO

If **L** and **S** are disjoint signatures then the wffs of USO(**L**) are all the strings of the form $\forall S\phi(x)$ where $\phi = \phi(x)$ is a first-order **LUS**-formula. If x has length n then for any **L**-structure D , for any $\bar{d} \in D^n$, $D \models \forall S\phi(\bar{d})$ if and only if every **LUS**-structure E

expanding D is a model of $\phi(\vec{d})$.

THEOREM 2.3

For all signatures L , USO(L) $\equiv$ PROV(L).

proof : $\forall S\phi(\vec{x})$ is equivalent to PROV({ $\forall \vec{x}(\phi(\vec{x}) \rightarrow P(\vec{x}))$ } ; P) and PROV(P;P) is equivalent to $\forall S(\wedge P \rightarrow P(\vec{x}))$.

Q.E.D.

We can bring a result of Fagin's in here to show that this means that there are PROV queries which are probably quite time consuming to answer. The reader unfamiliar with the basics of computational complexity as applied to query answering is referred to [Fag74]. There, Fagin describes how to code a finite structure in an acceptable form for an input to a Turing machine. A 0-ary query is polynomial time computable if and only if there exists a function f taking positive integers to positive integers such that f(n) is polynomial in n and there is a machine for answering the query which, on a structure of size n, answers the query in less than f(n) steps. A 0-ary query q is nondeterministically polynomial time computable if and only if there is such a function f and a not necessarily deterministic machine which can answer the query in less than f(n) steps. Fagin gives examples of queries which are of the latter type and probably not of the former type. Whether these queries actually are in one class and not the other is an open problem similar to the well known problem of whether NP=P or not (see [DaW83] or [GaJ79]).

Fagin in [Fag74] showed that the 0-ary queries which are nondeterministically polynomial time computable are precisely those which have negations in USO (the language of existentially quantified second order queries ESO contains exactly the negations of USO queries). Since the set of polynomial time computable queries is closed

under negation this means that there are probably PROV queries which take a long time to compute - i.e. are not polynomial time computable.

So PROV is very much a general framework as opposed to a language with practical applications.

2.5 Datalog as a sub-language of PROV

We have defined a very expressive query language, PROV, which is really too expressive to be practical - it can express queries which take a long time to compute. Let us now go to the other extreme and cut back on expressive power with the aim of being left with nice easy queries. By aiming for the class of provability queries which logic programming without function symbols can handle it is not surprising that we end up with Datalog.

A sub-language of PROV can be specified simply by defining a class **C** of theories and saying that PROV(**P**;P) is in the sub-language if and only if **P** is in **C**. To end up with those queries which logic programming can handle it seems a simple matter of using the class of Horn-clause theories as **C**.

However, a query language containing all wffs PROV(**P**;R) where **P** is a Horn-Clause theory (i.e. **P** consists entirely of Horn-Clauses) is not the same as one that will only use Horn-Clause theories when deducing answers using theorem-proving. For example, suppose **L**={L}, **S**={R,S},

P={ $\forall x$ R(x), $\forall x$ (R(x) $\rightarrow$ L(x)) },

and the **L**-structure D has only one element d such that D $\models \neg$L(d). The facts about D are $\forall x(x=d)$ and $\neg$L(d). We know that D$\models$PROV(**P**;**S**)(d) if and only if FA(D)$\cup$**P**$\vdash$S(d) if and only if

$\forall x R(x)$

$\forall x(R(x) \rightarrow L(x))$

$\forall x(x=d)$

$\neg L(d)$

$S(d)$

is a valid deduction, which it is. In all expansions of D to a model of **P**, S(d) is true because there are no such expansions. So we see that the defining theory, **P**, is Horn-Clause but that non-Horn-Clause deductions must be done to answer a query.

The problem here is that having a symbol from **L** in the head of a clause of our defining theory allows us to deduce a contradiction. In fact if we do not allow symbols from **L** in the heads of clauses (of the defining theory) then we never have this problem. Under these conditions we only ever need to use the positive atomic propositions in the facts about the **L**-structure in doing our deductions.

For example, suppose that **L**={L}, **S**={S,R},

P={ $\forall x$ S(x) , $\forall x$ (L(x) $\rightarrow$ R(x)) }

and the **L**-structure D has domain {a,b} with D$\models$L(a)/$\backslash\neg$L(b). Then D$\models$Prov(**P**;R)(a) because

$\forall x$ [(x=a) $\vee$ (x=b)] *

$a \neq b$ *

L(a)

$\neg L(b)$ *

$\forall x$ S(x)

$\forall x$ (L(x) $\rightarrow$ R(x))

R(a).

In fact we can carry through this proof whether the asterisked sentences are there or not. Similarly we can not prove R(b) from the axioms plus or minus the asterisked ones. Thus b is not a correct answer to PROV(P;R) asked of D.

This is the reason for our restriction on the heads of Datalog queries. This is enough to guarantee that we can use logic programming. PFACT is defined in section 2.3.

LEMMA 2.4

If **L** and **S** are disjoint signatures, **P** an **L**∪**S**-theory consisting wholly of Horn-clauses with only relations from **S** in their heads, R from **L** then
for all **L**-structures D, for all $\bar{d} \in D^n$,
D ⊨ PROV(P;R) $(\bar{d})$ if and only if
PFACT(D)∪**P** ⊢ $R(\bar{d})$.

proof:
The if half is easy as PFACT(D) is a subset of FA(D). To prove the converse we assume that D ⊨ PROV(P;R) $(\bar{d})$ and show that any **L**∪**S**∪D-structure E which is a model of PFACT(D) and **P**, also satisfies E⊨$R(\bar{d})$. Suppose we have such an E.

We define an **L**∪**S**-expansion F of D by putting F⊨$S(\bar{e})$ if and only if E⊨$S(\bar{e})$.

In fact, F is a model of **P** as we can discover by considering each clause of **P** individually. Assuming that F satisfies all the conjuncts in the body of a clause for some substitution $\bar{x}\backslash\bar{e}$ of the variables in the clause then any conjunct in the body is either
1) L($\bar{x}$) for L from **L**, whence $F \models L(\bar{x}(\bar{e}))$, so $D \models L(\bar{x}(\bar{e}))$, so $L(\bar{x}(\bar{e}))$ is in PFACT(D), so $E \models L(\bar{x}(\bar{e}))$; or
2) is S($\bar{x}$) for S from **S**, whence $F \models S(\bar{x}(\bar{e}))$, so by construction $E \models S(\bar{x}(\bar{e}))$.
Hence E satisfies the body with $\bar{x}$ replaced by $\bar{e}$ and E is a model of **P** and so of this clause so E satisfies the head under this substitution. By the construction of F so does it.

Hence F is a model of **P** expanding D and so by hypothesis $F \models R(\vec{d})$. By its construction so is E as required.

Q.E.D.

The other restrictions we made to logic programs to make them into Datalog queries were making them function and constant free. Since logic programming can "handle" functions and constants this might seem to have been an unnecessary restriction.

However, we know from [Hod85] that if we use logic programming with function and constant symbols then we end up being able to solve any first-order problem so we can answer any provability query. Since our purpose is to build up from Datalog to more expressive languages without being too expressive we have to avoid function symbols unless we want to reach the expressive power of PROV again.

What about just constants? Then the distinction between the constant symbols in PFACT(D) and those in **P** would lead to failure to implement the provability semantics. For example, consider the ({P},{R,c,Y})-theory $\mathbf{P} = \{ R(c) , \forall x (P(x) \wedge R(x) \rightarrow Y) \}$.
If the {P}-structure has domain {a} and $D \models P(a)$ then $D \models PROV(\mathbf{P};Y)$ under the extended provability semantics. This is because any expansion of D to a model E of **P** must have c interpreted as a so $E \models P(a) \wedge R(a)$ and thence $E \models Y$. On the other hand, if we query Y of the program

$P(a) \leftarrow$

$R(c) \leftarrow$

$Y \leftarrow P(x) , R(x)$

we do not get "yes". The program does not know that every expansion of D also satisfies $\forall x (x=a)$.

So, if we want to keep a simple semantics (provability), a simple implementation

(logic program consisting of description of structure followed by description of query) and not be too expresive then we stick to relational theories.

2.6 Definite Existential Implications

Can we use a practical extension of pure logic programming to help us answer queries from a different generalisation of Datalog? Lloyd and Topor in [LlT84] suggest making PROLOG more expressive by allowing any first-order formula in the body of clauses and give an implementation of the extension. However the implementation treats negation as failure. Let us see if we can follow their idea of allowing more in the bodies of clauses, but stay with a simple semantics and not make things too hard to implement.

Allowing equality in the body is no problem - $\forall xy$ ($x=a \rightarrow R(x,y)$) is equivalent to $\forall y\ R(a,y)$ and $\forall xyz$ ($x=y \rightarrow S(x,y,z)$) is equivalent to $\forall xz\ S(x,x,z)$. It is easy to see how to trivially rewrite a theory to take care of equality - there is no need to have an explicit ability to allow it in the bodies.

As for inequality, we will see below that allowing inequality in the bodies of clauses presents a similar problem to that of allowing negations of relation symbols from the extensional language.

We have conjunctions in the bodies of Horn-Clauses - what can be done about disjunctions? It is commonly done (see for example [LlT84]) and quite correct to replace $\forall x$ ($A(x) \vee B(x) \rightarrow C(x)$) by the two clauses $\forall x$ ($A(x) \rightarrow C(x)$) and $\forall x$ ($B(x) \rightarrow C(x)$). Again we don't need disjunctions explicitly in the bodies of clauses.

Allowing negation in the body of a clause seems to be the most popular extension of pure logic-programming. However this is usually not implemented as negation. Thus one can write a theorem with negation in it into the computer but not use the computer to

correctly deduce from the theorem. Rather, the ad-hoc rule negation as failure is used. We have already compared negation and NAF. We impose a strict separation between first-order theories from which we want to make deductions and the programs with NAF which might look like theories but are instructions for a computer to follow. If we want to make automated deductions from a first-order theory then the theory is one thing and the theory that looks like the program which we use to help us make the deductions is another thing.

We will make use of negation as failure in implementations but to suppose that we can use it to help us in deducing from theories containing unrestricted use of negation in the bodies of (generalised) clauses would be a mistake. A common problem occurs when we can not prove P(a), say, we assume $\neg$P(a) and then we use $\neg$P(a) to prove P(a) in some way. Such logic is obviously flawed. However this problem will never occur with atomic propositions P(σ) where P is from our extensional signature **L** if we continue to prohibit symbols from **L** from being in the heads of clauses. For in that case we can never use $\neg$P(σ) to prove P(σ). In fact, if we can not prove P(σ) then $\neg$P(σ) is actually in the facts about the **L**-structure. In this section we will investigate fully the use of negations of facts from the structure.

On the other hand, implementing negations of intensional relations directly as negation as failure will cause the problems referred to above. In fact it is very hard to handle such clauses. We can see this by noting that $\forall x$ (A(x) $\wedge$ $\neg$B(x) $\rightarrow$ C(x)) is logically equivalent to $\forall x$ (A(x) $\rightarrow$ B(x) $\vee$ C(x)). Thus, allowing negations in general in the bodies of clauses is equivalent to allowing disjunctions in the heads of clauses. Such a clause is called an *indefinite* clause by Gallaire, Minker and Nicolas in [GMN84a]. They note that it is very hard to deduce from theories containing indefinite clauses.

Our first extension of the basic Horn-Clause query language allows extensional negations and inequality in the bodies of sentences.

DEFINITION

if **L** and **S** are disjoint signatures then the **L**∪**S**-sentence $\forall \tau \ (\wedge_{i=1}^{n} A_i \rightarrow P(\sigma))$ is a *definite existential* (**L**,**S**)*-implication* if and only if

1) τ is a possibly empty tuple of distinct variable symbols;

2) n is a non-negative integer;

3) each A_i is either

3a) $R(\sigma)$ for R from **L**∪**S**,

3b) $\neg R(\sigma)$ for R from **L** or

3c) $u \neq v$;

and 4) P is a relation symbol from **S**.

DEFINITION if **L** and **S** are disjoint signatures then a finite **L**∪**S**-theory is called a definite existential (**L**,**S**)-theory if and only if it only contains definite existential (**L**,**S**)-implications.

Now we define the definite existential query language - definite as only one atomic formula appears in the head, existential as no universal quantifications appear in the conjuncts in the body.

DEFINITION If **L** is a signature then the existential sub query language DE(**L**) of PROV(**L**) consists of those queries PROV(**P**;R) from PROV(**L**) in which **P** is a definite existential (**L**,**S**)-theory, **S** is a signature disjoint from **L** and R is a symbol from **S**.

For example, of a {parent , married , male }-structure we can now ask who is an aunt of whom. If **P** is

{ $\forall$xyzu (parent(y,x) $\wedge$ parent(z,y) $\wedge$ parent(z,u) $\wedge$ $u \neq v$ $\wedge$ $\neg$male(u)
$\rightarrow$ aunt(u,x)),

$\forall$xyzuv (parent(y,x) $\wedge$ parent(z,y) $\wedge$ parent(z,u) $\wedge$ $u \neq v$ $\wedge$ male(u) $\wedge$ married(u,v)
$\rightarrow$ aunt(v,x)) }

then on any { parent , married , male }-structure D, D $\models$ PROV(**P**;P) (a,b) if and only if a is an aunt of b (given the obvious meaning in terms of the other relations).

Chandra and Harel in [ChH85] pointed out an obvious deficiency in the expressivity of their version of Datalog. They then remedied this by extending **H** to **H'** which is the same as DE. That there has been some improvement in expressive power by us making this move can be judged by using Chandra and Harel's example of a query which is not in Datalog but clearly in DE.

If **L** has just a 1-ary symbol R in it then the 1-ary **L**-query q given by a$\in$q(D) if and only if D $\models\neg$R(a), is called the complementation or negation of R. Clearly q is in DE(**L**).

To see that it is not in Datalog assume for contradiction that it is - say q = PROV(**P**;P) where **P** is a Horn-clause (**L**,**S**)-theory. Let D be the **L**-structure with domain {a} such that D $\models\neg$R(a). So D $\models$ PROV(**P**;P)(a) and by lemma 2.4, PFACT(D)$\cup$**P** $\vdash$ P(a). But PFACT(D) is empty so **P**$\vdash$P(a). Now if E is the **L**-structure with domain {a} such that E $\models$R(a) then a$\notin$q(D). However **P**$\vdash$P(a) so PFACT(E)$\cup$**P** $\vdash$ P(a) so E $\models$PROV(**P**;P)(a). This is the desired contradiction.

We know that to answer PROV(**P**;R)($\bar{a}$)? asked of a structure D, we can try to prove that R($\bar{a}$) follows from FA(D)$\cup$**P**. For Datalog **P** we can use a Horn-clause theorem-prover. We will see that if we treat negated extensional relations and inequality as just some more extensional relations then we can use the same theorem-prover for DE queries.

We have something to prove here : one can not always just regard negated relations as new relations. The deduction

$\forall x\ (\ S(x) \rightarrow A(x)\ )$

$\forall x\ (\ \neg S(x) \rightarrow A(x)\ )$

$\forall x\ A(x)$

is valid while

$\forall x\ (\ S(x) \rightarrow A(x)\ )$

$\forall x\ (\ R(x) \rightarrow A(x)\ )$

$\forall x\ A(x)$

is not. It requires the special circumstances of our definite existential theories to make similar replacements work. Consider the deduction

$\neg S(a)$

$\forall x\ (\ \neg S(x) \rightarrow A(x)\ )$

A(a)

which is valid as is

R(a)

$\forall x\ (\ R(x) \rightarrow A(x)\ )$

A(a).

In fact, we will show that if $\neg\ R$ is systematically replaced by a new relation symbol (R^*) in the axioms for the theorem-prover then the type of proofs that we are interested in still go through.

Let us suppose that L and S are disjoint signatures. We choose another signature L'

disjoint from both **L** and **S** such that :

1) for all n-ary R from **L** there is n-ary R^* in **L***;

2) there is a 2-ary relation symbol ineq in **L***;

3) all the R^*s and ineq are different.

Given a finite **L**-structure D we have defined FA(D) as

DCA(D) ∪ UNA(D) ∪ PFACT(D) ∪ NFACT(D).

Now we define

UNA^*(D) = $\bigwedge_{c \neq d \in D}$ ineq(c,d) ;

$NFACT^*$(D) = { $R^*(\bar{d})$ | $R \in$ **L**,$D \models \neg R(\bar{d})$ }.

We shall call the Horn-clause theory UNA^*(D) ∪ PFACT(D) ∪ $NFACT^*$(D), FA(D)*. Given an definite existential (**L**,**S**)-theory **P** we define **P*** as **P** with $u \neq v$ in bodies replaced by ineq(u,v) and $\neg R(\bar{x})$ replaced by $R^*(\bar{x})$.

LEMMA 2.5

If **L** and **S** are disjoint signatures , **P** is a definite existential (**L**,**S**)-theory, D is a finite **L**-structure, n is a non-negative integer, R is n-ary from **S** and $\bar{a} \in D^n$ then

for all n-ary R from **S**, for all $\bar{a} \in D^n$,

D $\models$ PROV(**P**;R) ($\bar{a}$) if and only if UNA^*(D)∪PFACT(D)∪$NFACT^*$(D)∪**P*** $\vdash R(\bar{a})$.

proof: (only if)

Suppose D $\models$ PROV(**P**;R) ($\bar{a}$) and that E is an **L**∪**L***∪**S**∪D-structure such that E is a model of FA(D)* ∪**P***. (D acts both as a finite **L**-structure and the set of constant symbols which its domain can be taken to be). We must prove that E is a model for $R(\bar{a})$.

Define the **L**∪**S**-structure F as follows: F is an expansion of D i.e. $F|_L$=D and for every n-ary S from **S** and n-tuple $\bar{d}$ from D, $F \models S(\bar{d})$ if and only if $E \models S(\bar{d})$. If we now show that F is a model of **P** then $F \models R(\bar{a})$ and we will have our desired result.

So take any clause c $\forall \bar{x}$ ($\bigwedge_{i=1}^{n} A_i(\bar{x}) \rightarrow P(\bar{u})$) from **P**. And suppose $\bar{e}$ from F satisfies $F \models \bigwedge_{i=1}^{n} A_i(\bar{e})$.

By going through each possibility for each A_i in turn it will be shown that $E \models A_i^*(\bar{e})$ where A_i^* is A_i with negation of an **L** relation changed to a positive appearance of the corresponding **L*** symbol and inequality changed to a use of *ineq*. But **P*** contains the clause c with just these changes made and E is a model of **P***. Thus $E \models P(\bar{u}(\bar{e}))$. No changes are made to the head of any clause when moving from **P** to **P***. But by the construction of F it must also be a model of $P(\bar{u}(\bar{e}))$ as it agrees with E on relations from **S**.

We know that $F \models A_i(\bar{e})$. Let us now show that $E \models A_i^*(\bar{e})$. There are five cases depending on the nature of A_i.

If A_i is $R_i(\bar{v}_i)$ with R_i from **L** then $F \models R_i(\bar{v}_i(\bar{e}))$. Since $F|_L$ is D then $D \models R_i(\bar{v}_i(\bar{e}))$. But then $R_i(\bar{v}_i(\bar{e}))$ would be in PFACT(D) which is a subset of $FA(D)^*$ of which E is a model. Finally since in this case A_i^* is A_i we have $E \models A_i^*(\bar{e})$.

If A_i is $\neg R_i(\bar{v}_i)$ with R_i from **L** then as above $D \models \neg R_i(\bar{v}_i(\bar{e}))$. In this case though we have a negative fact so $FA(D)^*$ contains $R_i^*(\bar{v}_i(\bar{e}))$ and again $E \models A_i^*(\bar{e})$.

If A_i is $x_p \neq x_q$ then $F \models e_p \neq e_q$ but F has the same domain as D so e_p and e_q are different elements of D. Thus $ineq(e_p, e_q)$ is in $FA(D)^*$ and so $E \models A_i^*(\bar{e})$.

If A_i is $S_i(\bar{v}_i)$ with S_i from **S** then the construction of F from E implies that E is also a model of $A_i^*(\bar{e})$.

This completes the only-if proof.

(if)

Suppose that $FA(\mathrm{D}) \cup \mathbf{P}^* \vdash R(\bar{\alpha})$. We must show that $\mathrm{D} \models \mathrm{PROV}(\mathbf{P};R)(\bar{\alpha})$. So given any $\mathbf{L} \cup \mathbf{S}$-structure F such that $F|_{\mathbf{L}}$ is D and F is a model for $\mathbf{P}$ we want to show that F is also a model of $R(\bar{\alpha})$. To do this we do the opposite of what we have just done and construct a model E of $FA(\mathrm{D}) \cup \mathbf{P}^*$ which agrees with F on $\mathbf{S}$.

We simply make the $\mathbf{L} \cup \mathbf{L}^* \cup \mathbf{S} \cup D$-structure E by expanding F by defining interpretations for *ineq* and each K^* for K in $\mathbf{L}$. We put $E \models ineq(c,d)$ if and only if c and d are different elements of D and $E \models K^*(\bar{d})$ whenever $F \models \neg K(\bar{d})$. It is straight forward to show that E is a model for $FA(\mathrm{D})^*$. A similar case by case argument to that in the first half of the proof shows that E is a model of $\mathbf{P}^*$. Thus $E \models R(\bar{\alpha})$ and by construction of F so is F.

Q.E.D.

In the light of [Hod85] this result might not seem worthy of mention: Hodges' theorems imply that if we are allowed to code the description of the queried structure then we can answer any provability queries using a Horn clause theorem prover. However, Hodges' coded theories make great and essential use of function symbols which puts this work out of the scope of this thesis: we are limiting ourselves to implementations which do not use function symbols. It should also be noted that the coded theories also make use of equality axioms whose ability to cause looping makes them unsuitable for depth-first implementations. On the other hand, our coding in the DE case does not use function symbols and does not necessarily introduce clauses like the equality axioms which are prone to causing looping. The programmer who is capable of avoiding looping problems in writing pure logic programs can use the same skill to try and keep them out of the coded Horn clause theories.

Thus DE is a practical language. But is it expressive enough?

2.7 DEF

All the queries in the language DE are easy to answer - at least as easy as those in Datalog. It is also quite expressive. However, as we are about to show, it does not even capture all the FO queries. We should be able to extend it to an even more expressive language without making the result too impractical. In fact, in this section we do just that, and end up with a language DEF, of definite queries, which we will spend a lot of the thesis investigating.

As Chandra and Harel [ChH85] show, their language **H**, which is equivalent to our DE, can not express universal quantification - i.e. if **L** contains 1-ary R then q, given by $a \in q(D)$ if and only if $D \models \forall x\, R(x)$, is not expressible in **H**.

To see that q is not in DE(**L**) suppose that we have D with domain {a} such that $D \models R(a)$ and E with domain {a,b} such that $E \models R(a)$ but $E \models \neg R(b)$. Then it is clear that $UNA'(D) \cup PFACT(D) \cup NFACT'(D)$ is a subset of $UNA'(E) \cup PFACT(E) \cup NFACT'(E)$. If q was in DE(**L**), say q = PROV(**P**;P) for **P** a DE (**L**,**S**)-theory then by the last theorem since $a \in q(D)$, $D \models PROV(\mathbf{P};P)(a)$, $UNA'(D) \cup PFACT(D) \cup NFACT'(D) \cup \mathbf{P}' \vdash P(a)$, $UNA'(E) \cup PFACT(E) \cup NFACT'(E) \cup \mathbf{P}' \vdash P(a)$, $E \models PROV(\mathbf{P};P)(a)$ and so $a \in q(E)$. This is clearly not so. So we have a contradiction.

After noting that universal quantification is missing from **H'**, Chandra and Harel go on to try and add a limited form of negation of intensional relation symbols to the bodies of implications. We shall look at the resulting language, **C**, later. For now, we do the perhaps more obvious thing and just add universal quantification to the bodies of sentences.

DEFINITION

If **L** and **S** are disjoint sentences then an **L**∪**S**-sentence ϕ is a *definite* (L,S)-*implication* if and only if $\phi = \forall x\ (\ \wedge_{i=1}^{n} A_i \rightarrow P(\tau)\ )$ where :

1) x is a possibly empty tuple of distinct variable symbols;

2) n is a non-negative integer;

3) each A_i is either

3a) $R_i(x_i)$ for R_i from **L**,

3b) $\neg R_i(x_i)$ for R_i from **L**,

3c) $u \neq v$

3d) $S_i(x_i)$ for S_i from **S** or

3e) $\forall y S_i(x_i)$ for S_i from **L**∪**S**;

4) P is from **S**.

DEFINITION

If **L** and **S** are disjoint signatures then an **L**∪**S**-theory **P** is a definite (L,S)-theory if and only if **P** is finite and consists only of (L,S)-implications.

DEFINITION

If **L** is a signature then DEF(**L**) the definite query language on **L**, is the sub-language of PROV(**L**) containing precisely those PROV(**P**;P) such that there is a signature **S** such that **P** is a definite (L,S)-theory.

It should be clear that the DCA(D) may be necessary in using FA(D)∪**P** to answer DEF PROV(**P**;P) on D. Thus we need something more than a Horn-clause theorem-prover. We shall investigate some possibilities later in the thesis.

One thing that is nice to know about DEF is that all definite theories are minimisable and hence intuitively pleasing to use to define queries (see section 2.2).

LEMMA 2.6

If **P** is definite then **P** is minimisable.

proof:

Suppose that **P** is a definite (**L**,**S**)-theory. Given any finite **L**-structure D we will expand it to a model of **P** in which the interpretation of each S from **S** contains precisely those elements which are in the interpretation of S in every expansion of D to a model of **P**. This is simply achieved by expanding D to E where, for each k-ary S from **S**, for each k-tuple $\bar{a}$ from D E⊨S($\bar{a}$) if and only if D ⊢ PROV(**P**,S)($\bar{a}^{\frown}$). It only remains for us to prove that E is in fact a model of **P**.

Suppose that $\phi = \forall \bar{x}(\bigwedge_{i=1}^{q} A_i \rightarrow A)$ is a sentence from **P** and that $\bar{a}$ is from D such that for each i=1...q , E is a model of $A_i(\bar{x}\backslash\bar{a})$. We must show that E is a model of $A(\bar{x}\backslash\bar{a})$.

If F is any expansion of D to a model of **P** then for each A_i either:

1) A_i is R($u\bar{x}$) for R from **L** whence E⊨R($u\bar{x}(\bar{a})$) so D⊨R($u\bar{x}(\bar{a})$) and so F⊨$A_i(\bar{x}\backslash\bar{a})$;

2) A_i is $u \neq v$ so E⊨$u(\bar{a}) \neq v(\bar{a})$ so D⊨$u(\bar{a}) \neq v(\bar{a})$ and so F⊨$u(\bar{a}) \neq v(\bar{a})$;

3) A_i is ¬R($u\bar{x}$) so E⊨¬R($u\bar{x}(\bar{a})$) so D⊨¬R($u\bar{x}(\bar{a})$) and so F⊨¬R($u\bar{x}(\bar{a})$);

4) A_i is S($u\bar{x}$) for S from **S** whence P(D)⊢S($u\bar{x}(\bar{a})$) so F⊨S($u\bar{x}(\bar{a})$) i.e. F⊨$A_i(\bar{x}\backslash\bar{a})$; or

5) A_i is $\forall y$S($u\bar{x}$) whence for all b from D, E⊨S($u\bar{x}(\bar{a},b)$) so as above F⊨S($u\bar{x}(\bar{a},b)$) and again E⊨$A_i(\bar{x}\backslash\bar{a})$.

Thus F⊨$\bigwedge_{i=1}^{q} A_i(\bar{x}\backslash\bar{a})$ but F⊨**P** so F⊨$A(\bar{x}\backslash\bar{a})$. Hence, if A is S($u\bar{x}$) then D⊨PROV(**P**;S)($u\bar{x}(\bar{x}\backslash\bar{a})$) and so E⊨S($u\bar{x}(\bar{x}\backslash\bar{a})$). Thus E is a model of ϕ and hence of **P**.

Q.E.D.

2.8 IMP

We have started with a very expressible language PROV , restricted our interest to a much less expressible but much more implementable language Datalog and then gradually added features to increase power again. Two different features have been considered:

1) allowing inequality and basic negations in bodies to get DE; and

2) adding universal quantification to get DEF ;

In fact we can show quite easily that by adding one more feature to definite theories we can again regain the expressive power of PROV. Whether DEF itself has this power also is a question which will have to wait a little longer to be answered. To define the language IMP of implicational queries we extend DEF by allowing multiple heads to our implications.

DEFINITION

If **L** and **S** are disjoint signatures then an **L**∪**S**-sentence ϕ is an (L,S)-implication if and only if $\phi = \forall \bar{x} \; (\; \bigwedge_{i=1}^{n} A_i \rightarrow \bigvee_{j=1}^{m} P_j(\bar{v}_j) \;)$ where:

1) $\bar{x}$ is a possibly empty tuple of distinct variable symbols;

2) n is a non-negative integer;

3) each A_i is either

3a) $R_i(\bar{u}_i)$ for R_i from **L**,

3b) $\neg R_i(\bar{u}_i)$ for R_i from **L**;

3c) $u \neq v$,

3d) $S_i(\bar{u}_i)$ for S_i from **S** or

3e) $\forall y S_i(\bar{u}_i)$ for S_i from **L**∪**S**;

4) m is a positive integer; and

5) each P_j is from S.

DEFINITION

A finite LUS-theory P consisting solely of (L,S)-implications is called an implicational (L,S)-theory.

DEFINITION

If L is a signature then IMP(L) is the sub-language of PROV(L) containing all the wffs PROV(P;P) such that P is an implicational (L,S)-theory, S is a disjoint signature from L and P is from S.

It is not too hard to show that we have regained the expressivity of PROV. Obviously IMP's queries are all expressible in PROV - they are all actually in PROV.

To show the converse we use a well known result to convert a given theory into an equivalent one in clausal form. Unfortunately the process adds some constant symbols - Skolem constants - which we then must get rid of. We must also cope with the possibility of clauses with empty heads being produced. It is not a difficult result to establish.

THEOREM 2.7

For all signatures L , PROV(L) $\equiv$ IMP(L).

proof: That IMP(L) $\subseteq$ PROV(L) is easy.

To prove PROV(L) $\subseteq$ IMP(L) we assume q $\in$ PROV(L) , say q is PROV(P;P) where P is an LUS-theory and P is n-ary from S, and rewrite q as an IMP query.

We shall say that a theory **Q** is clausal if and only if it consists solely of sentences of the form $\forall(\bigvee_{i=1}^{n} A_i)$ where each A_i is either an atomic formula or the negation of an atomic formula. In [Nil71] it is proven that for any **P** as above there exists a clausal **LUSUF**-theory **Q** for some set **F** of function and constant symbols such that any model of **P** can be expanded to a model of **Q** and the **LUS**-reduction of any model of **Q** is a model of **P**. From now on in this proof we will regard constant symbols as 0-ary function symbols.

Now we proceed to remove the constant and function symbols in **F** from **Q**. To do this we choose two new (n+1)-ary relation symbol R_f and S_f for each n-ary function symbol f in **F**. Let **R** be the signature of those new relation symbols. An **LUSUR**-theory equivalent to **Q** is constructed in two parts:

1) one, $\mathbf{T}_1$, says that R_f codes a function i.e. there is a function f such that $R_f(\vec{x},y)$ is true if and only if $f(\vec{x})=v$; and

2) the other $\mathbf{T}_2$ is **Q** with all references to f's replaced equivalently by R_f's.

To define $\mathbf{T}_1$ we include the following three sentences for each n-ary function symbol f from **F**. We take an n-tuple of distinct variable symbols $\vec{x}$ not including y nor z then the sentences are:

$\forall\vec{x}yz(\neg R_f(\vec{x},y) \vee \neg R_f(\vec{x},z) \vee \neg(y \neq z))$

$\forall\vec{x}yz(R_f(\vec{x},y) \vee S_f(\vec{x},y))$

$\forall\vec{x}(\neg\forall y S_f(\vec{x},y))$.

To define $\mathbf{T}_2$ we recursively apply the following operation to remove the appearances of symbols from **F** from **Q**. Suppose that after a certain number of applications of this operation we have a theory containing $\forall(\bigvee_{i=1}^{n} A_i)$. Suppose further that the sentence contains a symbol from **F**. Say that one of the A_i is $P(\vec{u})$ or $\neg P(\vec{u})$ and one of the u_j is $f(\vec{v})$ (or c) for f (or c) from **F**. We replace the sentence as follows. First replace u_j in A_i by z where z is a new variable symbol. Second, add the disjunct $\neg R_f(\vec{v},z)$ to the whole

disjunction with the new A_i replacing the old. Finally, we universally quantify z in the new sentence. We keep applying this transformation until we have got rid of all the symbols from **F**. This will eventually happen (as we are always removing appearances of symbols from **F**) and then we have our theory T_2.

It is not hard to see that as far as sentences of **LUS** are concerned $T_1 \cup T_2$ has the same consequences as **Q**.

Now we are almost done. We would like to rewrite $T_1 \cup T_2$ as an (**L**, **SUR**)-implicational theory by simply putting negative disjuncts on the left of an implication as positive conjuncts. There are two obstacles. First, equality. Disjuncts in the head which contain equality can be replaced by conjuncts in the body containing inequality. Now we choose a new symbol *eq* and add a clause $\forall x\ eq(x,x)$. Conjuncts, x=y, containing equality can be replaced by *eq*(x,y).

Second is the problem of empty heads - we might have clauses with no positive disjuncts. Solving this is a simple matter of choosing a new 0-ary relation symbol F say, adding it as a disjunct to any headless implications and also adding the sentences $\forall x(F \rightarrow R(x))$ for each symbol R. Its not hard to see that we have an equivalent theory.

Q.E.D.

Thus, in increasing expressivity from Datalog to DE to DEF to IMP we have gone from easy queries to compute to time consuming ones. In the rest of the thesis we will show that the change from a practical to an impractical language occurred when we moved from DEF to IMP.

Chapter 3

Fixed Points

In this chapter we determine the expressive power of DEF by showing how it compares with an established query language - namely FO+LFP. First, we lay some foundations for fixed point work then we show how to build the FO+LFP language on them. We next define a different fixed point language $\mathcal{LFP}$ which looks very much like our provability language DEF. By using and reworking a few results of other people we can conclude that $\mathcal{LFP}$ is equally expressible as FO+LFP. Finally we show that $\mathcal{LFP}$ is equivalent in expressive power to DEF. The corollary that FO+LFP $\equiv$ DEF allows us, amongst other things, to separate DEF from IMP and PROV in expressiveness. It also shows that DEF is a practical language with neither too weak an expressive power, so that it is deficient in practice, nor too strong a power so that it can express time consuming queries.

3.1 Least Fixed Points

Least fixed point languages are based on the well-known phenomena of the existence of least fixed points of monotone operators in complete partially ordered sets. We shall quickly look at the concepts involved and establish this fundamental fixed point theorem. The basics from algebra can be found, for example, in [Kur65], the fixed point theorem itself in [Tar52].

A set P with a binary relation $\leqslant$ called the *order* on P and written $p \leqslant q$ is called a

partially ordered set or *poset* , if and only if the following conditions are all satisfied:

$\forall p$ in P , $p \leqslant p$;

$\forall p$, q in P , if $p \leqslant q$ and $q \leqslant p$ then $p=q$;

$\forall p$, q and r in P , if $p \leqslant q$ and $q \leqslant r$ then $p \leqslant r$.

A simple example of a poset which is of interest to us is the set $REL_n(D)$ of n-ary relations on a finite set - or domain of a finite structure - D. We order $REL_n(D)$ by writing $\alpha \leqslant \beta$ if and only if for all $\bar{d}$ in α , $\bar{d}$ is also in β. The basic properties of set inclusion imply that $REL_n(D)$ is a poset.

If P is a poset then we define the concepts of *least upper bound* , *l.u.b.* , of a subset of P and *greatest lower bound* , *g.l.b.* , of a subset of P in the usual way. We will call P a *complete* poset if and only if each of its subsets have both a l.u.b. and a g.l.b.. $REL_n(D)$ is a complete poset - for example the l.u.b. of a set of relations is the join of them all.

A map from a set to itself will be called an *operator* on the set. If F is an operator on a poset P we can ask whether or not F preserves the ordering on P. F is called *monotone* if and only if for every p,q in P, if $p \leqslant q$ then $F(p) \leqslant F(q)$.

For example, if $\bar{a}$ is any n-tuple from D then we have an operator F defined on $REL_n(D)$ which takes any n-ary relation on D to that relation joined with the one containing only $\bar{a}$. This is clearly monotone.

In that example many elements of $REL_n(D)$ are untouched by the operation of F. They are called fixed points of F. In general if P is a poset , F an operator on P and $F(p)=p$ then we call p a *fixed point* of F.

Sometimes we have a least fixed point- one that is $\leqslant$ than any other. In our last example the relation which contains only $\bar{a}$ was the least fixed point.

In other situations there is no least fixed point- maybe there are no fixed points or maybe there are two which are incomparable by $\leqslant$. The operator $\alpha \rightarrow \neg\alpha$ has no fixed point on REL_n(D). The operator which takes α to α if α is a singleton and otherwise takes α to the full n-ary relation on D has no one least fixed point on REL_n(D) if D has more than one element. Note that both these operators are not monotone.

Tarski's theorem says that that last observation is not surprising.

THEOREM 3.1

Any monotone operator F on a complete poset P has a least fixed point. If 0 is the bottom element of P (i.e. the l.u.b. of the empty subset) and P is finite then there is n such that $F^n(0)$ is the least fixed point.

proof:

Suppose that F is a monotone operator on a complete poset P. Since P is complete the empty subset of P has a l.u.b. which is thus a minimum element of P - let us call it 0.

We define a sequence $a_0, a_1, \cdots$ as follows :

$a_0 = 0$;

and for all $n \geqslant 0$, $a_{n+1} = F(a_n)$.

Since 0 is the minimum element of P, $a_0 \leqslant a_1$ and since F is monotone we can prove by induction that for all n , $a_n \leqslant a_{n+1}$ i.e. the chain is (not necessarily strictly) increasing.

If P is finite as it will be in all our applications we can obviously find a fixed point c along this sequence (otherwise we would run out of elements). However its not hard to prove the result for not necessarily finite P so let us keep going.

We keep defining a_α for all ordinals whether we have reached a fixed point or not.

As before $a_{\alpha+1}$ is $F(a_\alpha)$ but if β is a limit ordinal then we define a_β to be the l.u.b. of all the preceding a_αs.

Now by using transfinite induction we can prove that the sequence is still increasing. The α to α+1 case is as before, the limit β case is obvious by the definition of a_β.

Since P is a set we must eventually run out of new elements for the sequence and then a_α equals a_γ for α less than γ say. Then $a_\alpha \leqslant F(a_\alpha)=a_{\alpha+1} \leqslant \cdots \leqslant a_\gamma=a_\alpha$. So $F(a_\alpha)=a_\alpha$ and we have a fixed point.

Suppose that c is another fixed point of F in P. We know that $0 \leqslant c$ so $a_1=F(0) \leqslant F(c)=c$. By transfinite induction we can prove that $a_\alpha \leqslant c$ and so a_α is the least fixed point of F.

If P is finite then α is finite and $a_\alpha = F^\alpha(0)$ by definition.

Q.E.D.

3.2 FO+LFP

Because we will need detailed knowledge of the construction of FO+LFP for an expressibility proof and because it is interesting to compare its construction with that of our new language $\overline{LFP}$ it will be worth the trouble to define FO+LFP in detail. We expand upon the outline given in [Gur84] and [GuS85].

If D is a structure and $REL_n(D)$ is the poset of n-ary relations on the domain of D then we have noted and it is not hard to see that $REL_n(D)$ is a complete poset. In particular this holds when n=0 and $REL_n(D)$ has two elements - false $\leqslant$ true.

$REL_n(D)$ will be the complete poset in which we are finding least fixed points. The operators which we will be using will be defined via first-order formulae. Just to get the

notation cleared up ready for doing this we will say that if L is a signature, D an L-structure, $P_1, \cdots, P_k$ are relation symbols not in L, $\phi = \phi(x)$ is a first-order $L \cup \{P_1, \cdots, P_k\}$-formula which we might write as $\phi(P_1, \cdots, P_k, x)$, a is a tuple of elements from D (of the same length as x) and $\rho_1, \cdots, \rho_k$ are relations on the domain of D of the same arities as $P_1, \cdots, P_k$ respectively then we write $D \models \phi(\rho_1, \cdots, \rho_k, a)$ (or more correctly $D \models \phi(P_1 \backslash \rho_1, \cdots, P_k \backslash \rho_k, x \backslash a)$) if and only if the following equivalent conditions are true:

1) the expansion of D to an $L \cup \{P_1, \cdots, P_k\}$-structure E by interpreting P_i in E as ρ_i makes $E \models \phi(a)$;

2) replacing $P_i(a)$ by $a \in \rho_i$ in ϕ makes $D \models \phi(a)$ with the obvious semantics.

This is very standard notation.

Now to define an operator on $REL_n(D)$ by a first-order formula we proceed as follows. Suppose that L is a signature, P is an n-ary relation symbol not in L which we call a *dummy* relation symbol , $\phi(P, x)$ is a first-order $L \cup \{P\}$-formula with $x = (x_1, \cdots, x_n)$. Then $F_\phi : REL_n(D) \to REL_n(D)$ maps $\rho \in REL_n(D)$ to $F_\phi(\rho) \in REL_n(D)$ given by $a \in F_\phi(\rho) \iff D \models \phi(\rho, a)$ for every a in D^n.

An example of such an operator on a { 1-ary E }-structure D is the E-difference F_ϕ on D defined via $\phi(x) = E(x) \wedge \neg P(x)$. An element a is in $F_\phi(\rho)$ if and only if it is in E (i.e. $D \models E(a)$) and not in ρ.

The really interesting operators are those defined by formulas ϕ in which the dummy relation symbol does not appear under an odd number of nestings of negations - we say it only appears *positively*. Using a relatively simple induction on the construction of ϕ we can show that positivity of P in ϕ has the following straight forward relation with the order on $REL_n(D)$:

LEMMA 3.2 (technical result 1)

If P is an n-ary relation symbol not in the signature **L**, D is an **L**-structure , P appears only positively in the first-order **L**$\cup${P}-formula $\phi(P,x)$ and $\sigma \leqslant \rho$ are k-ary relations on D then for all a in D,

$D \models \phi(P \backslash \sigma, a) \Rightarrow D \models \phi(P \backslash \rho, a)$.

The reason why operators defined by positive formulas are interesting should then be apparent . If P appears only positively in ϕ then $\sigma \leqslant \rho \Rightarrow F_\phi(\sigma) \leqslant F_\phi(\rho)$. For if $a \in F_\phi(\sigma)$ then $D \models \phi(\sigma, a)$ so $D \models \phi(\rho, a)$ and $a \in F_\phi(\rho)$. Thus , in this case , F_ϕ is a monotone operator.

The basic idea of the FO+LFP language is that we can make use of such an operator to define a query. Suppose that **L** is a signature and k-ary P not in **L** appears only positively in the first-order formula $\phi(P,x)$ with $x=(x_1, \cdots, x_k)$. Then we define the k-ary (FO+LFP) (**L**) query $LFP_{P,x}(\phi)$ by picking out on each **L**-structure D as the relation of correct answers the k-ary relation ρ which is the least fixed point of the F_ϕ operator. It has a least fixed point because it is monotone on the complete poset REL_k(D).

The notation $LFP_{P,x}(\phi)$ as above seems more complicated than is needed to do the job - why not just $LFP(\phi)$? This is because the actual language FO+LFP is more complicated - it allows nesting of LFP operators within other LFP operators and within first-order formulae. Different dummy relation symbols are used to construct different operators at different levels of nesting. The subscript P on $LFP_{P,x}$ is needed to indicate which dummy variable is used to define the operator which has least fixed point $LFP_{P,x}$.

The syntax of (FO+LFP) (**L**) is defined by finding a signature **P** disjoint from L, constructing some *pre-wffs* by adding into the usual rules for construction of wffs of FO(**L**$\cup$**P**) the LFP formation rule below which binds some occurrences of symbols from P

and then taking as wffs those pre-wffs which have no free occurrences of symbols from P.

LFP formation rule

if n is a non-negative integer, P n-ary from P, $x=(x_1, \cdots ,x_n)$ an n-tuple of distinct variable symbols, θ is a pre-wff, $a=(u_1, \cdots ,u_n)$ is an n-tuple of variable symbols and P does not appear negatively in θ then $\chi = [\ LFP_{Px}(\theta)\ ](a)$ is a pre-wff. χ has free variables those of θ leave out x and add those in a, it has positive free predicates those of θ leave out P and has the same negative free predicates as θ.

To define the semantics for the wffs of the new language we first define meanings for the pre-wffs. That is , if $\theta(P_1, \cdots ,P_k,x)$ is a pre-wff of FO+LFP we specify under what circumstances $D\models\theta(\rho_1, \cdots ,\rho_k,a)$. This we do in the usual way by induction on the construction of θ.

For example suppose that $\theta=\forall y\psi$ and ψ has free appearances of $P_1, \cdots ,P_l$ and free variables from x or y. Suppose that we have already defined the meaning of ψ under all substitutions. Then θ has free appearances of $P_1, \cdots ,P_l$ and free variables from x. We say $D\models\theta(P_1\backslash\rho_1, \cdots ,P_l\backslash\rho_l,x\backslash a)$ if and only if for every b from D , $D\models\psi(P_1\backslash\rho_1, \cdots ,P_l\backslash\rho_l,x\backslash a,y\backslash b)$.

The only new situation is when $\theta=LFP_{Px}(\phi)\ (a)$. Let us suppose that P is n-ary so $x=(x_1, \cdots ,x_n)$ are all different , $a=(u_1, \cdots ,u_n)$, we know the meaning of ϕ always , the free variables y of θ are those of ϕ minus x plus those in a and the free predicates $P_1, \cdots ,P_l$ in θ are those in ϕ except P. We must decide whether or not $D\models\theta(P_1\backslash\rho_1, \cdots ,P_l\backslash\rho_l,y\backslash a)$.

To decide this we define a map $F:REL_n(D)\rightarrow REL_n(D)$ given by $e\in F(\alpha)$ if and only if $D\models\phi(P_1\backslash\rho_1, \cdots ,P_l\backslash\rho_l,P\backslash\alpha,y\backslash a,x\backslash e)$. We note that we can define this map because , by

induction , we have already specified the meaning of ϕ. Since P appears only positively in $\phi(\rho_1, \cdots, \rho_l, P, \bar{x}, \bar{x})$ F is monotone on $REL_n(D)$ and has a least fixed point β in $REL_n(D)$.

We define $D \models LFP_{P,\bar{x}}(\phi(P_1 \backslash \rho_1, \cdots, P_l \backslash \rho_l, P, \bar{y} \backslash \bar{a}, \bar{x}))(\bar{a}(\bar{a}))$ if and only if $\bar{a}(\bar{a})$ is in β.

Its a simple matter to now carry over the meanings of pre-wffs to those pre-wffs which are wffs. We do not have to worry about substitutions for predicate variables in the wffs though because they have no free appearances of dummy symbols.

For example if L contains 2-ary parent then

$\theta(u,y) = \neg\, LFP_{A,x}(\, parent(x,y) \vee \exists z\, (\, parent(x,z) \wedge A(Z)\,)\,)(u)$

is a wff of (FO+LFP)(L) and if parent is interpreted by {(a,b)| a is a parent of b } then $D \models \theta(a,b)$ if and only if a is not an ancestor of b.

In [ChH82a], Chandra and Harel related FO+LFP to other query languages and classes of queries. One of the important results from our point of view is the following.

THEOREM 3.3 (from THEOREM 5.4 [ChH82a])

The class of queries represented by FO+LFP is strictly contained in the class of polynomial time computable queries.

3.3 Definition of *LFP*

In this section we will give a rigorous definition of the new language *LFP*. It is a simple non-nesting version of the simultaneous fixed point language developed in [Gai84] and also is based on something closely related to the type of operator appearing in [EmK76]. The basic properties of simultaneous fixed points are discussed in [Mos74]. The underlying partially ordered set in which we will be finding least fixed points now

consists of tuples of relations on a structure. We shall examine this idea first. As in the FO+LFP case we will then see how to define operators on our poset using first-order formulae. This time we will use tuples of formulae. As before we can restrict our formulae to ensure that a monotone operator is defined. We shall bring all these steps together to give an exact specification of the syntax and semantics of our new language.

Finally, several differences between the constructions of FO+LFP and $\overline{LFP}$ will be noted. One of the main ones is , of course, the simultaneous multiple character of $\overline{LFP}$. The other is the fact that we have not made a provision for nesting of $\overline{LFP}$ operations while FO+LFP allows nesting of LFP. This is because the notation for $\overline{LFP}$ looks complicated enough without nesting so we have risked expressive power for the sake of neatness.

the poset

What complete partially ordered set are we finding our new multiple fixed points in?

DEFINITION

If m is a positive integer , $\overline{m}=(m_1, \ldots, m_m)$ is an m-tuple of non-negative integers , L is any signature and D is an L-structure

then

1) an $\overline{m}$-ary relation on D is any m-tuple $\overline{\rho}=(\rho_1, \ldots, \rho_m)$ of relations ρ_i on the domain of D such that each ρ_i is an m_i-ary relation ;

2) $REL_{\overline{m}}(D)$ is a poset with elements all the $\overline{m}$-ary relations on D ; and

3) $REL_{\overline{m}}(D)$ is ordered by $\leqslant$ where for every $\overline{\sigma},\overline{\rho} \in REL_{\overline{m}}(D)$

$\overline{\sigma} \leqslant \overline{\rho}$ if and only if for every i=1...m $\sigma_i \leqslant \rho_i$ i.e. the extension of σ_i is included in the extension of ρ_i.

It is easy to see that $REL_{\overline{m}}(D)$ is a complete poset when D is finite.

It is important to note that a multiple relation is not just an ordinary relation with a big arity - they don't have an extension but a tuple of extensions.

We have said that $REL_m(D)$ is a complete poset. Its bottom element $\varnothing$ is the m-tuple $\varnothing=(\varnothing_1, \ldots, \varnothing_m)$ where $\varnothing_i$ is falsity if $m_i=0$ and otherwise is the empty m_i-ary relation on D.

defining multiple operators

Now that we have our complete poset the next step towards constructing a least fixed point language is to find a symbolic way of defining operators (especially monotone ones) on the poset. If m is a positive integer , $\overline{m}=(m_1, \cdots, m_m)$ an m-tuple of non-negative integers and **L** a signature then we want to be able to specify an operator $F:REL_m(D)\rightarrow REL_m(D)$ for any given **L**-structure D.

Any such function mapping into a set of m-tuples can be broken down into m simpler functions acting together. If $\rho\in REL_m(D)$ then $F(\rho)=(F_1(\rho), \ldots, F_m(\rho))\in REL_m(D)$ each $F_i:REL_m(D)\rightarrow REL_{m_i}(D)$. Each F_j maps a tuple of relations to a single simple relation and so could possibly have been defined by a first-order formula with some dummy relation variable symbols.

As in the FO+LFP case a first-order formula with symbols from **L** plus some dummy relation variable symbols can be used to represent a mapping on relations. Now we choose m different relation symbols $P_1, \ldots, P_m$ not in **L** with each P_i being m_i-ary. To represent a possible F_j we use a first-order $\mathbf{L}\cup\{P_1, \ldots, P_m\}$-formula ϕ_j with free variables from $\overline{x}_j=(x_{j1}, \ldots, x_{jm_j})$. Given $\rho\in REL_m(D)$ we define $F_j(\rho)\in REL_{m_j}(D)$ as follows:

$\forall\overline{a}\in D^{m_j}, \overline{a}\in F_j(\rho) \quad \Leftrightarrow \quad D\models\phi_j(P_1\backslash\rho_1, \ldots, P_m\backslash\rho_m, \overline{x}_j\backslash\overline{a})$.

Now it is a simple matter to put several of these maps together to define an operator.

DEFINITION Suppose we are dealing with L-structures. If m is a positive integer , $\bar{m}=(m_1, \ldots, m_m)$ an m-tuple of non-negative integers , $\mathbf{P}=\{P_1, \ldots, P_m\}$ is a signature disjoint from L such that each P_i is m_i-ary , $\phi_1, \ldots, \phi_m$ are LUP-formulae, and $\bar{x}_1, \ldots, \bar{x}_m$ are tuples of variable symbols each $\bar{x}_i=(x_{i1}, \ldots, x_{im_i})$ an m_i-tuple such that all the free variables from ϕ_i appear in $\bar{x}_i$

then the string $r=[\phi_1 \rightarrow P_1(\bar{x}_1), \ldots, \phi_m \rightarrow P_m(\bar{x}_m)]$ is called an $\bar{m}$-ary *generalised* L-*prescription* for multiple operators.

We call this string a "generalised prescription" because we are going to be solely interested in a special case for which we will reserve the name "prescription". Why is it a prescription for multiple operators?

DEFINITION If r is a generalised prescription as above then on any L-structure D we can use r to define an operator $F:REL_m(D) \rightarrow REL_m(D)$ as follows:

given $p \in REL_m(D)$, for all i=1...m , for all $\bar{a} \in D^{m_i}$,

$$\bar{a} \in [F(p)]_i \iff D \models \phi_i(P_1 \backslash \rho_1, \ldots, P_m \backslash \rho_m, \bar{x}_i \backslash \bar{a}).$$

We say that r *prescribes* F on D.

monotone operators

We saw in the construction of FO+LFP that the easiest way to ensure that a symbolically defined operator is monotone is to make the formula positive in the dummy variables. We shall follow this example in constructing $\overline{LFP}$. Things are even easier if we don't allow nesting of $\overline{LFP}$ operations . In this case any negations can be pushed into the atoms. In fact we wont suffer any loss in expressibility if negations are only allowed to appear on non-dummy atoms.

DEFINITION If **L** is a signature and $r=[\phi_1 \rightarrow P_1(x_1), \ldots, \phi_m \rightarrow P_m(x_m)]$ is a generalised L-prescription for multiple operators then we call r an **L-*prescription*** for multiple operators if and only if each ϕ_i is built

from only truth and falsity, equality and inequality, atoms from **L**, negated atoms from **L** and atoms from $P_1, \ldots, P_m$

using only conjunction , disjunction and universal or existential quantification.

A simple proof by induction on the construction of each ϕ_i will show that:

LEMMA 3.4 (technical result 2)

If r is an m-ary L-prescription for multiple operators, D is an L-structure and r defines F on D then F is a monotone operator on $REL_m(D)$.

the query language

Finally, we can define our new fixed point language:

Syntax of $\overline{LFP}$(L)

1) If $c \leqslant m$ are positive integers , $\bar{m}=(m_1, \ldots, m_m)$ is an m-tuple of non-negative integers and $r=[\phi_1 \rightarrow P_1(x_1), \ldots, \phi_m \rightarrow P_m(x_m)]$ is an $\bar{m}$-ary L-prescription for multiple operators then $\overline{LFP}(r;P_c)$ is a well formed formula of $\overline{LFP}$(L);

and 2) the only well formed formulas of $\overline{LFP}$(L) are those defined in 1).

Semantics of $\overline{LFP}$(L)

If $\overline{LFP}(r;P_c)$ is the well formed formula of $\overline{LFP}$(L) given in 1) above then it represents the m_c-ary L-query q defined as follows:

if D is any L-structure and r defines $F:REL_m(D)\to REL_m(D)$ on D then, by lemma 3.4, F is monotone on $REL_m(D)$ which is a complete poset and so, by theorem 3.1, F has a least fixed point $p\in REL_m(D)$. We say for all $\bar{a}\in D^{m_c}$,

$$\begin{aligned} & D\models \overline{LFP}(r;P_c)(\bar{a}) \\ \Longleftrightarrow\ & \bar{a}\in q(D) \\ \Longleftrightarrow\ & \bar{a}\in \rho_c. \end{aligned}$$

For example,

$\overline{LFP}$(mother(x,y) $\vee$ father(x,y) $\to P_1$(x,y) , P_1(x,y) $\vee \exists z$ (P_1(x,z) $\wedge P_2$(z,y)) $\to P_2$(x,y) ; P_2)

is just the ancestor query on any {mother,father}-structure. Suppose that D contains Richard, Mary, William and Stanley such that father(Richard , William), father(William , Stanley) and mother(Mary , William) are the only true atomic propositions in D. The prescription r as above prescribes F on D which behaves as follows.

$F(\emptyset,\emptyset)$ = ({ (Richard , William) , (William , Stanley) , (Mary , William) } , $\emptyset$).

$F^2(\emptyset,\emptyset)$ = ({ (Richard , William) , (William , Stanley) , (Mary , William) } ,
{ (Richard , William) , (William , Stanley) , (Mary , William) }).

$F^3(\emptyset,\emptyset)$ = ({ (Richard , William) , (William , Stanley) , (Mary , William) } ,
{ (Richard , William) , (William , Stanley) , (Mary , William) ,
(Richard , Stanley) , (Mary , Stanley) }).

$F^4(\emptyset,\emptyset) = F^3(\emptyset,\emptyset)$.

The answers to the query are the ordered pairs in the second set of the ordered pair $F^3(\emptyset,\emptyset)$ which is the least fixed point of F. These are the right answers to the ancestor query.

superficial comparisons

Now that we have a perfectly well defined query language what are we going to do

with it?

We are not going to develop it as a practical language. It may look much neater at expressing queries than FO+LFP but it also looks something like deductive database approaches to query posing. This is not surprising because the language has been defined to bridge the gap between deductive databases and conventional query languages. We will see in later chapters that a language as powerful as $\overline{LFP}$ can be easily implemented using existing theorem-proving methods - the importance of $\overline{LFP}$ will be as a standard of expressive power.

Let us compare the expressibility of FO+LFP and $\overline{LFP}$. The first step in such a comparison will be to see whether the constructions of the languages offer any clues to relative expressibility. We know that FO+LFP allows nesting of least fixed point operations while $\overline{LFP}$ does not. Negations can appear anywhere in FO+LFP queries provided they amount to an even number when nested above a dummy relation symbol. Negations only appear on non-dummy atoms in $\overline{LFP}$. On those points FO+LFP appears more powerful than $\overline{LFP}$. However we know that $\overline{LFP}$ has its multiple character while FO+LFP is based solely on simple operators. No easy conclusions can be drawn.

In fact its quite a job to compare the languages.

3.4 Expressing FO+LFP in $\overline{LFP}$

Our task is now to show that FO+LFP is (not necessarily strictly) less expressive than the new language $\overline{LFP}$. The essence of the problem is to get rid of the nesting of LFP operators which appear in FO+LFP queries. As we know $\overline{LFP}$ does not allow nesting of $\overline{LFP}$ operators.

Fortunately, most of the work has been done for us by Immerman in [Imm82]. He

showed that the nestings of LFP operators can be collapsed even inside FO+LFP itself. So we do not even need the extra power of multiple operators which $\overline{LFP}$ makes available for us - each FO+LFP query is equivalent to one with only one LFP operator.

In [ChH80b] Chandra and Harel laid the groundwork for Immerman's result. They defined a fixed point hierarchy of queries formed by applications of negation, quantification and LFP. We translate their notation into that of FO+LFP -

$\Sigma_0 = \{ \phi \mid \phi$ is a quantifier free FO query $\}$.

for each ordinal α,

$\Pi_\alpha = \{ \neg\phi \mid \phi$ is from $\Sigma_\alpha \}$,

$\Sigma_{\alpha+1} = \{ \exists x\phi \mid \phi$ is from $\Pi_\alpha \}$ and

for each limit ordinal β,

$\Sigma_\beta = \{ LFP_{P,x}(\phi)(u) \mid$ P only appears positively in ϕ, $\alpha<\beta$ and ϕ is from $\Sigma_\alpha \}$.

We follow Immerman in his modification of the definition of the hierarchy by allowing $LFP_{P,x}(\phi)(u)$ in Σ_β even when u is not simply a tuple of distinct variable symbols. However we stick with [ChH80b] on the question of positive versus monotone. Immerman allows $LFP_{P,x}(\phi)(u)$ in Σ_β when ϕ is monotone (on all structures - finite or infinite) in P. This causes no problems because Immerman's expressions are all equivalent to expressions already in FO+LFP ([Lyn59]).

Chandra and Harel show that the hierarchy quickly captures all of FO+LFP -

THEOREM 3.5 (THEOREM 4.4 [ChH80b])

$\cup_{\alpha<\omega^2}\Sigma_\alpha$ = FO+LFP.

Immerman's important result is that the hierarchy collapses at the first fixed point level -

THEOREM 3.6 (THEOREM 2 [Imm82])

Σ_ω = FO+LFP.

The basis of the proof is being able to express the negation of any query $LFP_{P,x}(\phi)(u)$ as a positive LFP query. To achieve this, Immerman defines some fixed point queries which answer questions about ranks of tuples - a rank of a tuple being the number n of times the monotone function F_ϕ must be applied to the empty relation before the tuple is in $F^n_\phi(\varnothing)$. We can express, in Σ_ω, when a tuple has the maximum finite rank of any tuple, and when a tuple has a greater rank than this. These latter tuples are the ones not in LFP(F_ϕ). Immerman gives a concrete demonstration of this construction by defining the negation of the ancestor query as $LFP_{S,x}(\phi)(u)$ for some first-order formula ϕ.

We can now prove our lemma -

LEMMA 3.7

For all signatures L, (FO+LFP) (L) $\subseteq \overline{LFP}$ (L).

proof:

Given any (FO+LFP) (L) query, Immerman's result implies that it is equivalent to a (FO+LFP) (L) query of the form $\psi(y) = LFP_{P,x}(\phi)(u)$ for $\phi(x) = \phi$ from FO (L$\cup${P}). Now we simply let r= [$\phi \rightarrow P(x)$, $P(u) \rightarrow Q(u)$]. It is not too hard to show that $LFP_{P,x}(\phi)(u)$ is equivalent to $\overline{LFP}(r;Q)$.

Q.E.D.

3.5 Expressing $\overline{LFP}$ in FO+LFP

In this section we will try to express any $\overline{LFP}$ query as a FO+LFP one. This would seem to be a simple matter of combining all fixed point relations produced by the $\overline{LFP}$ operator on a structure into one big LFP fixed point. Unfortunately, the simple approach does not quite work and we have to make things a bit more complicated. All the ideas are based on work in [ChH85] which, in turn, is related to the ideas on collapsing fixed points in [Mos74]. Immerman, in lemma 4.1 [Imm86], proves a simple case of our result.

Let us suppose that we have an $\bar{m}$-ary L-prescription r. We want to find a (FO+LFP) (L) query which is equivalent to $\overline{LFP}(\, r\, ;\, P_i\,)$ - i.e. gets the same answers on all finite L-structures. We could try coding each $\bar{m}$-ary relation $\bar{\rho}$ an any structure D, as a Σm_i-ary (simple) relation $\Lambda(\bar{\rho})$ given by $(\bar{a}_1, \cdots, \bar{a}_m) \in \Lambda(\bar{\rho})$ if and only if $\forall i\ \bar{a}_i \in \rho_i$. Then we could try coding r as a FO-formula ϕ in a corresponding way so that r prescribing F on D implies that ϕ prescribes Σm_i-ary f on D given by $f(\rho) = \Lambda(F(\sigma))$ where $\Lambda\sigma=\rho$. This simple idea can not work because $\bar{\rho} \rightarrow \Lambda(\bar{\rho})$ is not an injective coding - if any ρ_i is empty then $\Lambda(\bar{\rho})$ will be. It is not easy to overcome this problem, especially when the set of i such that the ith component of $F^n(\varnothing)$ is empty (where F is prescribed by r on D) depends on D and n.

We must consider an alternative approach. Chandra and Harel solved a similar problem in [ChH85] when they expressed any DATALOG (H in their notation) query in FO+LFP. They also start off trying to code multiple relations as simple ones - but in a different way. We adapt their idea to suit our purposes. We end up with the following coding of a prescription by a FO+LFP expression.

Suppose that L is a signature, $s = [\ \psi_1 \rightarrow P_1(\bar{x}_1), \cdots, \psi_m \rightarrow P_m(\bar{x}_m)\ ]$ is an $\bar{m}$-ary L-prescription, $\bar{m} = (\, m_1, \cdots, m_m\,)$, e is $[\log_2(m\,)]+1$, M is the maximum of the m_i and l is an integer between 1 and m. We proceed to define a (FO + LFP) (L) query s_l^*. First we

choose an M-tuple x and an e-tuple y of new variable symbols.

Let $code_i^l$ be the formula $(y_1=y_0)\wedge \cdots \wedge(y_e=y_0)$. For each i = 1 ... m different from l, let b_j be the jth bit in the binary representation $b_1b_2 \cdots b_e$ of i. Put ϵ_j equal to $y_l=y_0$ if b_j is 0 and equal to $y_l\neq y_0$ if b_j is 1. Now $code_i^l$ is the formula $\epsilon_1\wedge \cdots \wedge\epsilon_e$. For example, $code_3^5 = (y_1=y_0) \wedge (y_2\neq y_0) \wedge (y_3\neq y_0)$ - corresponding to the representation 011 for 3.

Now we choose a new M+e -ary relation symbol P. For each i = 1 ... m, we get ϕ_i by replacing any appearance $P_k(\alpha)$ of any P_k in ψ_i by

$\exists xy[code_k^l\wedge(x_1=u_1)\wedge \cdots \wedge(x_{m_k}=u_{m_k})\wedge P(x,y)]$.

Now let χ_i be $code_i^l\wedge\exists x_i[(x_1=x_{i1})\wedge \cdots \wedge(x_{m_i}=x_{im_i})\wedge\phi_i]$. Finally we put $\chi = \bigvee_{i=1}^{m}\chi_i$ and $s_i^*(x_1, \cdots ,x_{m_i}) = \exists x_{m_i+1} \cdots x_m y_0[LFP_{P,xy}(\chi)[x,y_0, \cdots ,y_0]]$.

For example, if s = [$P_2(x)\rightarrow P_1(x)$, $R(x)\rightarrow P_2(x)$] then

$s_1^*=\exists y_0[LFP_{P,xy}(\chi)(x,y_0,y_0,y_0)]$ where

$$\chi = [\ (y_1=y_0) \wedge (y_2=y_0) \wedge \exists u[\ (x=u) \wedge \phi_1\ ]\ ] \vee$$
$$[\ (y_1\neq y_0) \wedge (y_2=y_0) \wedge \exists u[\ (x=u) \wedge R(\alpha)\ ]\ ]$$

and where ϕ_1 is $\exists xy_0y_1y_2[\ (y_1\neq y_0) \wedge (y_2=y_0) \wedge (x=u) \wedge P(x,y_0,y_1,y_2)\ ]$.

For the coding to be successful we would want s_i^* to be equivalent to $LFP(s;\ P_i)$. We can prove this preliminary result.

LEMMA 3.8 (technical result 4)

For all finite L-structures D, for all $\bar{d}$ from D, if $D\models s_i^*(\bar{d})$ then $D\models LFP(s;P_i)(\bar{d})$.

Proving the converse, however, runs into trouble because the coding makes such use of the inequality relation on the domain of D that it falls down when the domain is

trivial - i.e. only has one element. For example, suppose that d is the lone element in the domain of an $\{R\}$-structure D, that $D \models R(d)$ and that

$s = [\, P_2(x) \rightarrow P_1(x)\,,\ R(x) \rightarrow P_2(x)\,]$

as above. Then $D \models \overline{LFP}(s;P_1)(d)$. However, we do not have $D \models s_1^{\uparrow}(d)$ because the least fixed point of the operator prescribed by $\chi \rightarrow P(x,y_0,y_1,y_2)$ is the empty 4-ary relation on D.

If D is non-trivial then we can use the coding.

LEMMA 3.9 (technical result 6)

For all non-trivial finite L-structures D, for all $\bar{d}$ from D, if $D \models \overline{LFP}(s;P_i)(\bar{d})$ then $D \models s_i^{*}(\bar{d})$.

If D is trivial then the coding may miss some of the tuples which are in the multiple fixed point but we can show that we catch those tuples which we know are in the multiple fixed point after one operation of the multiple operator.

LEMMA 3.10 (technical result 7)

For all finite L-structures D, if s prescribes G on D and $\bar{d} \in [G(\varnothing)]_{P_i}$ then $D \models s_i^{*}(\bar{d})$.

Chandra and Harel's solution to the problem of trivial databases is based on their version of this result. If we have a prescription r* which, on trivial databases, prescribes operators which get to their fixed point after one operation then the coding of r* will work. Thus, given any prescription r, if we can find such an r* which always prescribes operators with the same fixed point as r then it will give an equivalent FO+LFP query to the original one involving r.

If r = [$\phi_1 \to P_1(x_1), \cdots, \phi_m \to P_m(x_m)$] is an m-ary L-prescription and $\bar{m}$ an m-tuple then we define the m-ary L-prescription r* as follows. For any L-prescription s = [$\psi_1 \to P_1(x_1), \cdots, \psi_k \to P_k(x_k)$], $app(s)$ = [$app(\psi_1) \to P_1(x_1), \cdots, app(\psi_k) \to P_k(x_k)$] where $app(\psi_i)$ is got from ψ_i by replacing any appearance $P_q(\bar{x})$ of any P_q by ($\psi_q(x_q \backslash \bar{x}) \vee P_q(\bar{x})$). We put r* = app^{n-1}(r).

It should be clear that the following relationship holds.

LEMMA 3.11 (technical result 10)

If r prescribes F on D and r* prescribes G on D then F and G have the same least fixed point.

The important point for us as we try to overcome the problem of trivial databases is the following observation about any prescription.

LEMMA 3.12 (technical result 11)

If r prescribes F on trivial D then $F^m(\varnothing)$ is the least fixed point of F on D.

Now we can put all these results about r* and s_i^* together to prove the following result.

LEMMA 3.13

If L is a signature then for every $\overline{LFP}$(L) query there is an equivalent (FO+LFP)(L) query.

proof:

Suppose that m is a positive integer, π is an m-tuple of non-negative integers, r is the π-ary L-prescription [$\phi_1 \rightarrow P_1(x_1), \cdots, \phi_m \rightarrow P_m(x_m)$] and l is between 1 and m. We will show that $\overline{LFP}$(r ; P_l) is equivalent to $(r^*)_l^+$.

If D $\models (r^*)_l^+(\vec{d})$ then lemma 3.8 implies that D $\models \overline{LFP}(r^*;P_l)(\vec{d})$. However r* and r prescribe operators with the same fixed point so we also have D $\models \overline{LFP}(r;P_l)(\vec{d})$.

In proving the converse we identify two cases. If D is non-trivial and D $\models \overline{LFP}(r;P_l)(\vec{d})$ then we know that we also have D $\models \overline{LFP}(r^*;P_l)(\vec{d})$. Lemma 3.9 implies that D $\models (r^*)_l^+(\vec{d})$ as required.

If D is trivial and r prescribes F on D, then r* prescribes $G = F^m$ on D with $G(\varnothing)$ being the least fixed point of F. So, if D $\models \overline{LFP}(r;P_l)(\vec{d})$ then $\vec{d} \in [G(\varnothing)]_{P_l}$ and so lemma 3.10 implies that D $\models (r^*)_l^+(\vec{d})$.

Q.E.D.

Now we can put our results of the last two sections together to get a major result.

LEMMA 3.14

FO+LFP $\equiv \overline{LFP}$

3.6 FO+LFP versus DEF

In this section we give some quick proofs of the obvious connections between $\overline{LFP}(\mathbf{L})$ and DEF(L). It should be already clear that they are equally expressive. Here we just have to make some adjustments to turn the theory $\mathbf{P}_r$ which an L-prescription r looks like into a definite theory.

First we establish the equivalence - in terms of queries represented - between the prescription r and the theory $\mathbf{P}_r$. Then we show how to construct a prescription r from any definite theory P so that PROV(P;R) = PROV($\mathbf{P}_r$;R). We then show how to construct a definite theory P from any prescription r so that PROV(P; P_l) = PROV($\mathbf{P}_r$; P_l). Finally we sum up the results.

If L is a signature and $r = [\phi_1 \rightarrow P_1(x_1), \cdots, \phi_k \rightarrow P_k(x_k)]$ is an L-prescription then we define a $\mathbf{L} \cup \{P_1, \cdots, P_k\}$-theory $\mathbf{P}_r$ as $\{ \forall x_1(\phi_1 \rightarrow P_1(x_1)), \ldots, \forall x_k(\phi_k \rightarrow P_k(x_k)) \}$.

LEMMA 3.15

For each $l = 1 \ldots k$, PROV($\mathbf{P}_r$; P_l) = $\overline{LFP}(r;P_l)$.

proof:

part one) $D \models$ PROV($\mathbf{P}_r$;P_l)(c) implies $D \models \overline{LFP}(r;P_l)(c)$.

Suppose that $D \models$ PROV($\mathbf{P}_r$;P_l)(c). Let S be $\{P_1, \cdots, P_k\}$. We define an expansion E of D to an L∪S-structure by $E \models P_i(c)$ if and only if $D \models \overline{LFP}(r;P_i)(c)$. If we show that E is a model of $\mathbf{P}_r$ then we are done.

We show that E is a model of $\mathbf{P}_r$ by showing that if it is a model of any $\phi_i(x_i \backslash c)$ then it is a model of $P_i(c)$. Let π be the least fixed point of the operator F prescribed by r in D. Then if E is a model of $\phi_i(x_i \backslash c)$ we have $D \models \phi_i(\pi, x_i \backslash c)$. By the definition of F, $c \in [F(\pi)]_{P_i}$.

Thus $c \in [\pi]_{P_i}$. So D⊨$\overline{LFP}(r;P_i)(c)$ and so E is a model of $P_i(c)$ as required.

part two) D⊨$\overline{LFP}(r;P_i)$ implies D⊨PROV($\mathbf{P}_r$;P_i)(a).

Suppose that D⊨$\overline{LFP}(r;P_i)$ and that r prescribes F on D. Given any expansion of D to a model E of $\mathbf{P}_r$ we must show that E⊨$P_i(a)$. First we show by induction that for each n, for each e, for each i= 1 ... k, if $e \in [F^n(\varnothing)]_{P_i}$ then E ⊨$P_i(e)$.

It is trivially true for n=0.

Assume that it is true for n. Now if $e \in [F^{n+1}(\varnothing)]_{P_i}$ then D⊨$\phi_i(F^n(\varnothing),e)$. But each $[F^n(\varnothing)]_{P_j}$ is (by the inductive hypothesis) less than or equal to the interpretation P_j^E for P_j in E in the usual order on relations on the domain of D. Since ϕ_i is monotone with respect to the interpretations for the P_j, D⊨$\phi_i(P_1^E, \cdots, P_k^E, e)$. That is E ⊨$\phi_i(e)$. Since E is a model of $\mathbf{P}_r$ we have E ⊨$P_i(e)$ as required.

We know that for large enough n, $F^n(\varnothing)$ is the least fixed point π of F. We know that $a \in [\pi]_{P_i}$ so, using our inductively proved result for large enough n we have E ⊨$P_i(a)$ as required.

Q.E.D. (lemma 3.15)

LEMMA 3.16

If **L** and **S** are disjoint signatures, R is from **S** and $\mathbf{P}$ is a definite (**L**,**S**)-theory then there exists an L-prescription r such that PROV($\mathbf{P}_r$;R) = PROV($\mathbf{P}$;R).

proof:

part one) definition of r

Suppose that S is $\{P_1, \cdots, P_k\}$. For each P_i from S, if P_i is p-ary then choose distinct variable symbols $x_i=(x_{i1}, \cdots, x_{ip})$ which do not appear in P. We define an $L \cup \{P_1, \cdots, P_k\}$-formula ϕ_i as follows.

If P_i appears in the heads of no sentences in P then ϕ_i is *falsity*.

If P_i appears in the heads of precisely sentences $c_1, \cdots, c_m$ from P where m is non-zero then we put $\phi_i = \bigvee_{j=1}^{m} \beta_j$ where if c_j is $\forall y(\bigwedge_{i=1}^{n} A_i \rightarrow P_i(v))$ then β_j is $\exists y(\bigwedge_{i=1}^{p}(x_i = v_i) \wedge \bigwedge_{i=1}^{n} A_i)$.

Finally we put $r = [\phi_1 \rightarrow P_1(x_1), \cdots, \phi_k \rightarrow P_k(x_k)]$.

part two) $D \models PROV(P; R)(\bar{a})$ implies $D \models PROV(P_r; R)(a)$

Suppose that E is an expansion of D to a model of P_r. We will show that E is a model of P.

If $\forall y(\bigwedge_{i=1}^{n} A_i \rightarrow P_i(v))$ is any clause from P then it was called c_j, the jth clause containing P_i in the head in our construction of r. If c is from D such that $E \models \bigwedge_{i=1}^{n} A_i(y \backslash c)$ then $E \models \beta_j(x_i \backslash v(y \backslash c))$ so $E \models \phi_i(x_i \backslash v(y \backslash c))$. But $E \models P_r$ so E is a model of $P_i(v(y \backslash c))$ as required.

part three) $D \models PROV(P_r; R)(a)$ implies $D \models PROV(P; R)(\bar{a})$

Suppose that E is an expansion of D to a model of P. We will show that E is a model of P_r and hence E will be a model of $R(a)$ as required.

Suppose that E is a model of $\phi_i(x_i \backslash b)$. ϕ_i is not falsity (if E is a model of it) so say it is $\bigvee_{j=1}^{m} \beta_j$. Thus E is a model of one of the $\beta_j(x_i \backslash b)$ - say the one corresponding to the sentence $\forall y(\bigwedge_{i=1}^{n} A_i \rightarrow P_i(v))$ from P. By definition of β_j there must be a tuple c from D such that $E \models b = v(y \backslash c)$ and $E \models \bigwedge_{i=1}^{n} A_i(y \backslash c)$. But E is a model of P so E is a model of $P_i(v(y \backslash c))$. Thus $E \models P_i(b)$ as required.

Q.E.D. (lemma 3.16)

LEMMA 3.17

If L is a signature and $r = [\phi_1 \rightarrow P_1(x_1), \cdots, \phi_k \rightarrow P_k(x_k)]$ is an L-prescription then there exists a signature S containing $\{P_1, \cdots, P_k\}$ and a definite (L,S)-theory P such that for each $l = 1 \ldots k$, $PROV(P;P_l) = PROV(P_r; P_r)$.

proof:

part one) definition of S and P

We put each P_l in S and for each subformula ψ of each ϕ_l we order the free variables from ψ in some way $y_1, \cdots, y_m$ and put a new and different m-ary relation symbol S_ψ in S. That is all we put in S.

Before we construct P we will define an operation * on the set of those subformulas. We define $*(\psi)$ by induction on the construction of ψ.

$*(truth) = \{ S_{truth} \}$

$*(falsity) = \{ \}$

$*(u=v) = \{ \forall x S_{u=v}(x,x) \}$

$*(u \neq v) = \{ \forall xy(x \neq y \rightarrow S_{u \neq v}(x,y)) \}$

For each R from L, if the free variables in $R(x)$ were ordered y then

$*(R(x)) = \{ \forall y(R(x) \rightarrow S_{R(x)}(y)) \}$.

For each R from L, if the free variables in $\neg R(x)$ were ordered y then

$*(\neg R(x)) = \{ \forall y(\neg R(x) \rightarrow S_{\neg R(x)}(y)) \}$.

For each $i = 1 \ldots k$, if the free variables in $P_i(x)$ were ordered y then

$*(P_i(x)) = \{ \forall y(P_i(x) \rightarrow S_{P_i(x)}(y)) \}$

If the free variables from ϕ, ψ and $\phi \vee \psi$ were ordered x, y and z respectively then

$*(\phi \vee \psi) = \{ \forall y(S_\phi(x) \rightarrow S_{\phi \vee \psi}(z)), \forall y(S_\psi(y) \rightarrow S_{\phi \vee \psi}(z)) \} \cup *(\phi) \cup *(\psi)$.

If the free variables from ϕ, ψ and $\phi\wedge\psi$ were ordered x , y and z respectively then

$$*(\phi\wedge\psi) = \{\forall y(S_\phi(x)\wedge S_\psi(y)\rightarrow S_{\phi\wedge\psi}(z))\} \cup *(\phi) \cup *(\psi).$$

If x and y are the free variables in ϕ and $\forall u\phi$ respectively then

$$*(\forall u\phi) = \{\forall y(\forall u S_\phi(x)\rightarrow S_{\forall u\phi}(y))\} \cup *(\phi).$$

If x and y are the free variables in ϕ and $\exists u\phi$ respectively then

$$*(\exists u\phi) = \{\forall y u(S_\phi(x)\rightarrow S_{\exists u\phi}(y))\} \cup *(\phi).$$

Now for each i = 1 ... k if the free variables in ϕ_i were ordered y_i then we put

$P_i = \{\forall x_i(S_{\phi_i}(y_i)\rightarrow P_i(x_i))\} \cup *(\phi_i)$.

Finally we put $P = \cup_{i=1}^{k} P_i$.

It is easy to check that P is in fact a definite (L,S)-theory.

part two) $D\vdash PROV(P;P_i)(\sigma)$ implies $D\vdash PROV(P_r;P_i)(\sigma)$

Suppose that E is an expansion of D to a model of P_r. We expand E to an LUS-structure F as follows. If ψ is a subformula of one of the ϕ_i with its free variables ordered y then we put $F\vdash S_\psi(\sigma)$ if and only if $E\vdash\psi(y\backslash\sigma)$.

It is clear that F is a model of all the $*(\phi)$. If $F\vdash S_{\phi_i}(y_i(x_i\backslash\sigma))$ then $E\vdash\phi_i(y_i(x_i\backslash\sigma))$. But E is a model of P so $E\vdash P_i(\sigma)$ and hence $F\vdash P_i(\sigma)$. Thus F is a model of P.

Since $D\vdash PROV(P;P_i)(\sigma)$ F is a model of $P_i(\sigma)$ and so E is.

part three) $D\vdash PROV(P_r;P_i)(\sigma)$ implies $D\vdash PROV(P;P_i)(\sigma)$

Suppose that F is an expansion of D to a model of P. We prove that the restriction E of F to $L\cup\{P_1, \cdots, P_k\}$ is a model of P_r.

By induction on the construction of a subformula ψ it is easy to show that, because F is a model of each $*(\phi)$, if $E\vdash\psi(y\backslash\sigma)$ then $F\vdash S_\psi(\sigma)$.

If $\delta = y_i(x_i \backslash c)$ and $E \models \phi_i(\delta)$ then $F \models S_{\phi_i}(\delta)$ so $F \models P_i(c)$ and so $E \models P_i(c)$. Thus E is a model of P_r.

Since $D \models PROV(P_r; P_i)(\sigma)$ we have $E \models P_i(\sigma)$ and so $F \models P_i(\sigma)$ as required.

Q.E.D. (lemma 3.17)

LEMMA 3.18

$LFP \equiv$ DEF

proof:

Just combine the lemmas in the obvious way.

Q.E.D. (lemma 3.18)

Combining this with theorem 3.14 we arrive at the following.

THEOREM 3.19

DEF $\equiv$ FO+LFP

This allows us to finally answer our main unanswered question about the relative expressive powers of provability languages - we can now place DEF strictly between DE and IMP in expressive power.

COROLLARY 3.20

IMP represents queries which are not in DEF.

proof:

Let q be any one of the 0-ary polynomial time computable queries which is not in FO+LFP. Such queries exist by theorem 3.3. Since $\neg q$ will be non-deterministically polynomial time computable it will, by Fagin's result (THEOREM 6, [Fag74]), be in ESO and hence, by THEOREM 2.3, q is in PROV. Thus, by THEOREM 2.7, q is in IMP. Since q is not in FO+LFP it is not in DEF.

Q.E.D.

Chapter 4

Negation as Failure

In this chapter we will lay the foundations for our work on query languages based on logic programming with negation as failure. The starting point for such work must be a workable definition of negation as failure. We have seen that NAF can not be introduced in a declarative way so we have no choice but to look at the details of the query evaluation process. Here we define a version of such a process and compare it with the original version so we can write about practical results in future chapters.

4.1 A query evaluation process for Negation As Failure

In this section we define a version of the query evaluation process for negation as failure. We choose to define a general version not limited to any particular selection rule. However, the NAF rule we use is a fairly simple and incomplete one. We shall compare our QEP with other versions in the next section. We work without function symbols because they are not necessary for our purposes.

Programs are built from program clauses which are built from literals. These are positive literals and negative literals. A *positive literal* is an atomic formula "R($\bar{x}$)" where R is an n-ary relation symbol and $\bar{x}$ an n-tuple of variable symbols and/or constant symbols. A *negative literal* is "not (R($\bar{x}$))" where R($\bar{x}$) is a positive literal.

A *program clause* is anything of the form A $\leftarrow A_1, \cdots, A_q$ where A is a positive literal, q is a non-negative integer and each A_i is a literal. The *head* of the clause is A and

$A_1, \ldots, A_q$ is its *body* which may be empty.

If L is a signature and C a finite set of constant symbols then a *program* from LUC is a finite list of program clauses : each one containing only relation symbols from L and constant symbols from C.

In the context of logic programming a *query* is anything of the form $L_1, \cdots, L_p$ where p is a positive integer and each L_i is a literal.

A *goal* from LUC is anything of the form $\leftarrow L_1, \cdots, L_p$ where p is a non-negative integer and each L_i is a literal with relation symbol from L and constants from C. When p=0 , the goal , "$\leftarrow$", is called the empty goal.

We have seen that in the pure Horn-clause SLD-resolution case we construct a path of goals leading from the query goal to the empty goal. In this case it gives us a proof of inconsistency. In the NAF case we still try to construct such a path but the rule governing what is a valid step in the path is generalised to take account of negative literals. The relation between the path and a proof of inconsistency is no longer there.

The attempted construction of a path of goals proceeds by the selection of a literal from the last goal and, in the case that this is a positive literal, the choice of a clause of the program which says something about this literal (i.e. which has a head containing the same relation symbol). In SLD-resolution and most versions of NAF a rule governing the selection of literals is specified. The steps in paths of these versions must involve the literal indicated by the rule being used to get the next step. In our, more general version, a path can have its steps generated by the selection of any literal from one step to get to the next. We will say more about the relationship between our NAF and other NAFs in the next section.

Our use of the property of some goals being able in a finite amount of time to be

shown not to succeed means that we have to be more specific in getting from one step to the next possible one. We do not want to have to check for the failure of each of the extensions of a path got by trying any of the infinite number of possible mgus for a particular literal and a clause head. We suppose that we have a deterministic way of getting mgus - i.e. if G is a goal, L is a literal from G and A, which unifies with L, is the head of a clause c then we suppose that we have specified a pair of substitutions θ , ϕ as *the* mgus of L from G and A from c such that

1) θ is a substitution of the variables appearing in G;

2) ϕ is a substitution of the variables in c;

3) if θ' is a substitution of the variables in G and ϕ' is a substitution of the variables in c such that $L\theta'=A\phi'$ then there is a substitution α of the variables in $G\theta$ and in $c\phi$ such that $\theta\alpha=\theta'$ and $\phi\alpha=\phi'$.

Such a strong condition is necessary. For example, suppose we are not so fussy and allow θ taking x→x and y→y and ϕ taking y→x and x→y to be the mgus of Q(x) from $\leftarrow R(x,y),Q(x)$ and Q(y) from $Q(y)\leftarrow P(x,y)$. We should succeed Y from the program

Y ← Q(x) , R(x,y)

Q(y) ← P(x,y)

R(a,b) ←

P(c,a) ←

but the only hopeful path is

← Y

← Q(x) , R(x,y)

← P(y,x) , R(y,y)

← R(a,c)

which ends in failure. In fact, our definition prevents us making such a mistake by disallowing such θ and ϕ (they are not allowed because condition 3 does not hold with θ' taking x to x and y to y and ϕ' taking y to x and z to z).

In the SLD case we use unification to get the next step after a positive literal is selected. When the program and/or query contain "not" we must must specify how to continue a path after we select a negative literal from the previous step. Suppose we choose not(P(t)). Roughly, Clark's idea in [Cla78] was to see whether P(t) succeeds or fails and then if P(t) succeeds then do not continue the path via this literal and if P(t) fails then allow the next step in the path to be the same as the last step without not(P(t)). It might seem that we are capturing the closed world intuition by behaving as if "not(P(t)) ←" appears as a program clause whenever P(t) fails. In fact, the situation is more complicated.

There are at least three problems with this simple idea of negation as failure. First, how do we know whether P(t) fails? Many goals have infinite paths and these same goals can also have a successful path or alternatively no successful path at all. If the goal has an infinite path and no successful path then the computer can go off constructing the infinite path and never be able to conclude finally that there is no successful path. In practice we have to use the result of finite failure - i.e. the goal has a finite set of finite paths none ending in the empty goal. As we shall see, this distinguishes NAF from negation as not provable.

Second is the problem of variable symbols in P(t). If R(x) ← not(P(x)) appears in a program and P(x) succeeds with substitution taking x to a then are we really justified in failing R(x) according to our vaguely defined motivation for negation as failure? Suppose that P(b) fails. Shouldn't we succeed R(x) with the substitution taking x to b? Even trying all the possible substitutions θ for x and succeeding R(x) with θ if P(xθ) fails is not good enough because perhaps there are other elements in a model not named by constants which do not satisfy P. Do the implicit assumptions of the closed world assumption include one that there are no other elements apart from those named by constants which actually appear in the program? Even if we include this assumption we have already

made a lot more work for the computer - try $P(\pi\phi)$ for each substitution ϕ. Most common versions of PROLOG do not even try to capture the intuition correctly and use a very badly behaved implementation. It is better for the process to give up visibly when things get too hard for it rather than for it to carry on incorrectly. Clark [Cla78] first introduced a version of NAF which gave up , or floundered, if a non-ground literal was chosen. He also suggested an improved version which could handle some non-ground literals and the hard to implement extended version mentioned above. We will base our version on Clark's original one.

Third is the problem of negation loops. To find whether $P(\pi)$ succeeds or fails we might come to a point in a path where not($P(\pi)$) is selected. The computer will happily get stuck in a loop in such situations. When we are defining the concept of a path we must decide whether we have a path or not in such cases. For example, if

$P(a) \leftarrow$ not($P(a)$)

appears in the program then we must say whether ($\leftarrow P(a)$, $\leftarrow$ not($P(a)$) , $\leftarrow$) is a path or not. If we are not careful then we might say that it is a path if and only if it is not a path. We shall avoid this nonsense making circularity by attempting a definition inductively involving successive levels of "not"s.

DEFINITION

We define inductively on the level n the concepts of

1) an n-path for a goal G,

2) a closed set of n-paths for G,

3) $P(\pi)$ succeeds at level n with substitution θ, and

4) $P(\pi)$ finitely fails at level n.

All these concepts are relative to a fixed program with symbols from a fixed signature and finite fixed set of constant symbols.

Now suppose that n is a non-negative integer such that for all non-negative integers m less than n we have defined m-paths, closed sets of m-paths, success at level m and finite failure at level m.

n-paths for a goal G

We say that H is *got* from G *via* θ at level n if and only if

G is non-empty, it is $\leftarrow L_1, \cdots, L_p$ say, and there is an L_i such that either

GV1) L_i is positive, it unifies with the head A of a clause $A \leftarrow A_1, \cdots, A_q$ with the mgus θ , ϕ of L_i from G and A from the clause and H is $\leftarrow L_1\theta, \cdots, L_{i-1}\theta, A_1\phi, \cdots, A_q\phi, L_{i+1}\theta, \cdots, L_p\theta$

or

GV2) L_i is not($P(\bar{x})$), n is greater than zero, $P(\bar{x})$ is ground and finitely fails at level n-1, θ is the identity and H is $\leftarrow L_1, \cdots, L_{i-1}, L_{i+1}, \cdots, L_p$.

A sequence $(G_0, \cdots, G_k)$ of goals with a sequence $(\theta_0, \cdots, \theta_{k-1})$ of substitutions is an n-*path* for a goal G if and only if

1) G_0 is G

and

2) each G_{i+1} is got from G_i via θ_i at level n-1.

Closed sets of n-paths for a goal G

A set **G** of n-paths for G is *closed* if and only if

CS1) (G) , along with the empty list of substitutions , is in **G**,

and

CS2) if $(G, G_1, \cdots, G_k)$ along with $(\theta_0, \cdots, \theta_{k-1})$ is in G then either

CS2.1) G_k is empty

or

CS2.2) G_k is non-empty, it is $\leftarrow L_1, \cdots, L_p$ say, and there is an L_l such that

CS2.2.1) L_l is positive, and if it unifies with the head of a program clause

$A \leftarrow A_1, \cdots, A_q$ with mgus θ, ϕ of L_l from G_k and A

from the clause then

$(G, G_1, \cdots, G_k,$

$\leftarrow L_1\theta, \cdots, L_{l-1}\theta, A_1\phi, \cdots, A_q\phi, L_{l+1}\theta, \cdots, L_p\theta)$

along with $(\theta_0, \cdots, \theta_{k-1}, \theta)$ is in G

or

CS2.2.2) L_l is not($P(\pi)$), n is greater than zero, $P(\pi)$ is ground and either

CS2.2.2.1) $P(\pi)$ succeeds at level n-1

or

CS2.2.2.2) $P(\pi)$ finitely fails at level n-1 and

$(G, G_1, \cdots, G_k, \leftarrow L_1, \cdots, L_{l-1}, L_{l+1}, \cdots, L_p)$ is in G.

Success at level n

We say that $P(\pi)$ *succeeds* at level n with substitution θ if and only if there is an n-path for $\leftarrow P(\pi)$ ending in the empty goal, $\theta_0, \cdots, \theta_k$ is the sequence of substitutions in the path and θ is the restriction of the composition $\theta_0 \cdots \theta_k$ to the variables appearing in π.

Finite failure at level n

We say that P(π) *finitely fails* at level n if and only if there is a closed set of n-paths for $\leftarrow P(\pi)$ such that none end in the empty goal and there is a finite bound on their lengths. We call such a set a *closed bounded set of unsuccessful paths* for $\leftarrow$ P(π).

Success, Finite Failure and Non-Termination

We say that P(π) succeeds with substitution θ if and only if P(π) succeeds with substitution θ at some level. We say that P(π) finitely fails if it does so at some level. If P(π) neither succeeds nor finitely fails at any level then we say that it does not terminate. In the next section we will identify one particular type of non termination as being particularly annoying.

Before looking at a few examples let us prove a useful lemma.

LEMMA 4.1

If P(π) succeeds then P(π) does not finitely fail.

proof:

If P(π) succeeds at level k and finitely fails at level l then it is easy to see that it will both succeed and finitely fail at level m , the maximum of k and l. If **G** is a closed bounded set of unsuccessful m-paths for $\leftarrow P(\pi)$ and **P** is a successful m-path for $\leftarrow P(\pi)$ then it is easy to get a contradiction by showing (by showing by induction that equivalent paths to each of the sub-paths of **P** are in **G**) that an m-path the same as **P** (up to equivalence) is in **G**. By equivalence here we mean the same up to substitution under an invertible substitution.

Q.E.D.

Example 1

R(y) ← P(x,y) , Q(x)

P(x,x) ←

Q(a) ←

In this program a is a constant symbol and x and y are variable symbols as they usually are. R(x) succeeds at level 0 with answer x → a because the following is a 0-path for ←*R*(y). (← R(y) , ← P(x,y) , Q(x) , ← Q(x) , ←) with unifiers taking y to y to x to a respectively. We note that here and in other examples we will assume particular mgus as "the" mgus of literals.

Example 2

R(x) ← not(P(x))

P(a) ←

We suppose that this program comes from a finite set of constant symbols with a and b in it. Then { (← P(b)) } is a closed set of 0-paths for ← P(b) which is bounded and unsuccessful. Thus P(b) finitely fails at level 0. So (← R(b) , ← not(P(b)) , ←) is a 1-path for ← R(b) and R(b) is successful at level 1.

Example 3

P(a) ← not(P(a))

P(a) neither succeeds nor finitely fails at any level n as can be shown in a simple induction.

4.2 Implementing the QEP

In the last section we defined a query evaluation process which we called NAF. However, NAF is a term in common usage amongst logic programmers. What justification do we have for associating our concept with this term? Let us look briefly at what usually goes under the name and at the relationship between those concepts and our QEP.

The differing versions of logic programming with negation as failure can be classified by two independent criteria -

1) what, if any, literal selection rule do they use to continue paths and

2) what happens when a negative literal is selected.

In the literature, one can find several different approaches to the second problem. Our QEP behaves, when a negative literal is chosen to continue a path, as the original query evaluation process (qep) of Clark in [Cla78] does when its fixed selection rule selects a negative literal. Mu-Prolog (see [Nai82]) also behaves in the same way when its special selection rule selects a negative literal. This approach is quite a practical one (i.e. its easy to implement) and this fact, along with the fact that it was Clark's original one, are the reasons why it is used in this thesis. However, it does have the draw back of getting stuck - i.e. *floundering* - when a goal full of non-ground literals presents itself so that such a literal must be chosen.

Most of the other approaches which are mentioned here are attempts at reducing the amount of floundering which a qep does. Commercial Prolog versions usually use a flounder-proof rule which is not very amenable to theoretical considerations (see [Llo84] page 81). IC-Prolog (see [CMcC79]) uses a version which flounders less than ours and can be justified to some extent (as much as ours) on the basis of the closed world assumption with finite failure but it is not much referred to in the literature. Clark in [Cla78] used the same justification for an extended version of his original qep which does not flounder

but is only of theoretical interest because it is not implementable. Shepherdson [She85] makes some changes to this on theoretical grounds but again the version is not of practical use. We will leave it to other researchers to relate the results in this thesis to the behaviour of these other versions of NAF.

The other main way in which other versions of NAF might differ from ours is in the selection rule used, if any. Our QEP does not use one. Remember that a path is extended - i.e. a new goal H got from an old one G - by the selection of a literal L from the old goal such that either L is positive and H is the result of unifying L from G with the head of a clause or L is negative and ground , its positive part finitely fails and H is G without L. See GV1 from section 4.1 for details. We also specify a closed set by including all possible extensions of each path using one of the literals in its last goal (see CS2.2). In most versions of NAF a selection rule is fixed which specifies exactly which literal from the goal is to be used to extend paths in GV1 and CS2.2.

A selection rule is a map which picks out one literal from each non-empty goal. If R is a selection rule then we will talk about R-qep in which successful and finitely failing goals are those in which the choice of literals in GV1 and CS2.2 is that chosen by R. Sometimes, the term selection rule is used to mean a more general map whose choice of literal, from a goal in a path, is also dependent on what the initial query was and how long the path is. Shepherdson, in [She84], calls our kind of selection rule the narrow kind but, in [She85], he shows that such selection rules get all the answers.

It is clear that

RESULT 4.2

Assuming that QEP and R-qep use the same mgus, if a goal succeeds with a substitution (respectively finitely fails) at level n in R-qep then it succeeds with that substitution (respectively finitely fails) at level n in QEP.

However, it is easy to see that the converse does not hold. For example, consider the following program

$P \leftarrow P , Q .$

Suppose that **R** is the selection rule which picks P from the goal $\leftarrow P , Q$. Obviously, P finitely fails in QEP. However, **R**-qep gets stuck in a loop.

We have some work to do to justify our use of QEP on practical grounds. It is not necessarily of much use for us to get results about a qep which is far stronger than any commonly considered qeps. Our QEP gets all the results which any **R**-qep gets. We have seen that at least some **R**-qeps are not as strong. It would be nice if there was a maximal selection rule - i.e. a rule **R'** such that **R'**-qep gets any answer (i.e. any yes or no answer) which any **R**-qep does. It is easy to see that such a rule is impossible (consider which literal it picks from $\leftarrow P , Q$).

Although we can not find a universal maximal selection rule we can find a maximal one for each program (its a bit of work to show that this is so as we see below). In fact, not only is it maximal but it reaches the strength of our QEP. This is the result which enables us to relate results about our QEP back to results about versions of Clark's qep with particular selection rules.

It must be pointed out that the same result can be deduced fairly easily from theorem 5 [She84] and theorem 7.5 [She85]. One only needs to note that Shepherdson's definition of a selection rule is wide enough so that any path in our QEP is also a path using one of these selection rules. We justify presenting our proof on the basis that we will make use of all of its parts in the next chapter in proving our main results. It is readily conceded that investigations into the relationship between the behaviour of the NAF qep's and selection rules are exhausted by the results in [She84] and [She85].

Given a program PROG we define its maximal selection rule **R** as follows. All

reference to success and failure and other terms involved in a qep are referring to the behaviour of the goal in our QEP. For each goal G, if G does not finitely fail and contains at least one positive or ground negative literal then we get **R** to pick one of those literals. If G does not finitely fail but contains all non-ground negative literals then we pick any one of them. If G does finitely fail then the specification of **R** is more difficult. We find the least m such that G finitely fails at level m and the least l such that there is a closed bounded set of unsuccessful m-paths for G with all their lengths less than l. From one such set **G**, we note which literals from G are used to fail G or extend it into a path in G in accordance with the definition of a closed set. We get **R** to pick one of these literal from G.

Now we must justify our use of the term maximal in describing that selection rule.

THEOREM 4.3

If our QEP succeeds a goal G with substitution θ from program PROG then the version, **R**-qep, of NAF using the maximal selection rule for PROG will succeed $G\theta$ under an invertible substitution

and

if our QEP finitely fails a goal G from PROG then **R**-qep does too.

proof:

By induction on n we prove that

a) if QEP succeeds G under θ at level n then **R**-qep succeeds $G\theta$ under invertible ψ at level n; and

b) if QEP finitely fails G at level n and n is the minimum such integer then G finitely fails at level n in **R**-qep.

Assume that we have shown a and b up to n-1. Now for n.

PART a)

Suppose that the QEP succeeds G under θ at level n.

To assist us in this proof we will employ the concept of a path tree.

DEFINITION

If G is the goal $\leftarrow L_1, \cdots, L_p$ and n a non-negative integer then an n-*path tree* for G is a tree with each node labelled by a string called its *literal label* (*llabel*) and a string called its *clause label* (*clabel*) such that

for each node t of the tree one of the following is true

PT1) t is the root, both its llabel and its clabel are "*", t has exactly p successors $t_1, \cdots, t_p$ and each t_i is llabelled L_i ;

PT2) t has no successors, is llabelled with a literal and is clabelled "unfinished" ;

PT3) t is llabelled with a positive literal, $P(\bar{x})$ say, t is clabelled with a clause $P(\bar{y}) \leftarrow A_1, \cdots, A_q$ from PROG, there is a substitution ϕ of the variables in the clause such that $\bar{y}\phi = \bar{x}$, t has exactly q successors $s_1, \cdots, s_q$ and each s_i is labelled $A_i\phi$; or

PT4) t is llabelled with a ground negative literal, not($P(\bar{x})$) say, $P(\bar{x})$ finitely fails at level n-1 from PROG, t has no successors and is clabelled "negative".

Path trees will be used to represent a whole set of paths - each different only in the selection of literals from goals. We will call each such path an example of the path tree.

DEFINITION

An n-path $(G_0, \cdots, G_l)$ with $(\theta_0, \cdots, \theta_{l-1})$ is an *example* of an n-path tree for H if and only if there is a substitution ϵ and for each $i = 0 \ldots l$, there is a map α_i associating each appearance of a literal in G_i with a node in the tree such that

EG1) $H = G_0\theta_0 \cdots \theta_{l-1}\epsilon$;

EG2) if G_l is $\leftarrow K_1, \cdots, K_r$ then α_l maps K_j to a node llabelled $K_j\theta_l \cdots \theta_{l-1}\epsilon$; and

EG3) if K_s is chosen from G_l (as above) to get to G_{l+1} and $\alpha_l(K_j)=t$ then either

EG3a) K_s is positive,

t is clabelled with $A \leftarrow A_1, \cdots, A_q$,

the mgus of K_s from G_l and A from the clause are θ_l , ϕ_l,

G_{l+1} is $\leftarrow K_1\theta_l, \cdots, K_{s-1}\theta_l, A_1\phi_l, \cdots, A_q\phi_l, K_{s+1}\theta_l, \cdots, K_r\theta_l$,

α_{l+1} maps $K_u\theta_l$ to $\alpha_l(K_u)$

and α_{l+1} maps $A_u\phi_l$ to the successor s_u of t with llabel $A_u\phi_l\theta_{l+1} \cdots \theta_{l-1}\epsilon$ or

EG3b) K_s is negative and ground,

θ_l is the identity,

G_{l+1} is $\leftarrow K_1\theta_l, \cdots, K_{s-1}\theta_l, K_{s+1}\theta_l, \cdots, K_r\theta_l$,

the clabel of t is "negative"

and α_{l+1} maps $K_u\theta_l$ to $\alpha(K_u)$.

We will also find it useful to have an ordering on these path trees.

DEFINITION

We say that a tree H is a *sub-tree* of a tree G if and only if

there is a map β from the nodes of H to the nodes of G and a substitution γ such that for all nodes t of H,

ST1) $\beta(root_H)=root_G$,

ST2) if t has exactly q >0 successors in H then $\beta(t)$ has exactly q successors in G and β takes t's successors in a one-to-one manner to $\beta(t)$'s.

ST3) if $\beta(t)$ has a successor $\beta(u)$ in G then u is a successor of t in H,

ST4) if t is clabelled "*" then $\beta(t)$ is too, if t is clabelled with a clause then $\beta(t)$ is clabelled with the same clause and if t is clabelled "negative" then so is $\beta(t)$, and

ST5) if t is llabelled with a literal L then $\beta(t)$ is llabelled $L\gamma$.

Now back to the proof.

Since G succeeds under θ in QEP at level n, there is an n-path ($G_0, \cdots, G_l$) with ($\theta_0, \cdots, \theta_{l-1}$) such that $G_0 = G$, G_l is empty and $\theta=\theta_0 \cdots \theta_{l-1}$. By lemma 4.4 we can construct an n-path tree G for $G\theta$ which is finished - i.e. has no nodes clabelled "unfinished" - and of which the path is an example.

By lemma 4.5, we can use G to construct an n-path ($H_0, \cdots, H_l$) with ($\psi_0, \cdots, \psi_{l-1}$) for $H_0 = G\theta$ such that it uses the selection rule R, it is an example of G and either H_l is all non-ground negative literals, or the path is successful, or it can be extended further.

By lemma 4.6 H_l is not all non-ground negative literals so either we get a successful path or we can keep extending it.

To show that it is impossible for us to keep getting longer and longer such paths we note that, by lemma 4.7, if a path tree H is constructed from H-path (as in lemma 4.4) then it will be a sub-tree of G . But, by lemma 4.4, each step of H-path causes the number of nodes in the constructed tree H to increase or stay the same and if the number of nodes stays the same then the number of unfinished nodes decreases. So for long enough path we get more nodes in the constructed H than in the path tree G of which H is a sub-tree. This is impossible by the definition of sub-tree.

So suppose that we have a maximal such path - so it is successful. Since it is an example of G, $G\theta = H_0\psi_0 \cdots \psi_{l-1}\epsilon$ for some ϵ. If $\psi=\psi_0 \cdots \psi_{l-1}$ then we have $G\theta = G\theta\psi\epsilon$ so ψ is invertible on $G\theta$ as required.

It remains to prove the lemmas.

LEMMA 4.4

Given an n-path, G-path = $(G_0, \cdots, G_l)$ with $(\theta_0, \cdots, \theta_{l-1})$, we can construct an n-path tree G for $G_0\theta_0 \cdots \theta_{l-1}$ such that

1) G-path is an example of G with ϵ being the identity,

2) the range of α_l is the unfinished nodes of G, and

3) if we extend G-path by one more step and construct a new tree then it has as many or more nodes and if it has the same number then it has less unfinished nodes.

proof:

If $G_0 = \leftarrow L_1, \cdots, L_p$ then we start with a tree, G_0, with a root with p successors $t_1, \cdots, t_p$. We give the root llabel and clabel "*" and each t_i llabel $L_1\theta_0 \cdots \theta_{l-1}$ and clabel "unfinished". If we put $\epsilon = \theta_0 \cdots \theta_{l-1}$ and α_0 mapping each L_i to t_i then it is easy to see that the path (G_0) is an example of the tree G_0.

We keep constructing trees $G_0, G_1, \ldots, G_l$, and maps $\alpha_0, \cdots, \alpha_l$ such that, for each i,

C1) $(G_0, \cdots, G_l)$ with $(\theta_0, \cdots, \theta_{l-1})$ is an example of G_l with $\alpha_0, \cdots, \alpha_l$ being the appropriate maps and $\epsilon = \theta_l \cdots \theta_{l-1}$ being the appropriate substitution and

C2) the range of α_l is the unfinished nodes of G_l.

These conditions hold when i=0. Suppose that they hold for i and we are moving on to i+1. Say G_l is $\leftarrow K_1, \cdots, K_r$ and K_s is chosen to get to G_{l+1}. Say that $\alpha_l(K_s) = t$. There are two cases.

case 1: K_s is positive. Say that G_{l+1} is got by the unification of K_s and the head of the clause $A \leftarrow A_1, \cdots, A_q$. Suppose that θ_l, ϕ_l are the mgus of K_s from G_l and A from the clause. To construct G_{l+1} from G_l we take off t's "unfinished" clabel and give it the clause as its clabel, we give t exactly q successors $s_1, \cdots, s_q$, we llabel each s_u with

$A_u\phi_l\theta_{l+1} \cdots \theta_{l-1}$ and we clabel s_u "unfinished". To see that the conditions are satisfied define $\alpha_{l+1}(K_u\theta_l)=\alpha_l(K_u\theta_l)$, put $\alpha_{l+1}(A_u\phi_l)=s_u$ and put $\epsilon=\theta_{l+1} \cdots \theta_{l-1}$.

case 2: K_s is negative. Take off t's "unfinished" clabel and reclabel it "negative". Define $\alpha_{l+1}(K_u\theta_l)=\alpha_l(K_u)$.

We note that in the end, when i=l, then if the path was successful and so G_l is empty, by C2 the tree is finished - i.e. has no unfinished labels. It is also easy to check that each extension of G_l to G_{l+1} does not decrease the number of nodes and if it does not increase them then it decreases the number of unfinished nodes instead.

Q.E.D. (lemma 4.4)

LEMMA 4.5

Given an n-path tree G and a selection rule S which selects positive or ground negative literals if there are any then we can construct an n-path $(H_0, \cdots ,H_l)$ with $(\psi_0, \cdots ,\psi_{l-1})$ such that

1) the path proceeds by the selection of literals according to S,

2) it is an example of G,

3) if G is finished then either

3.1) H_l is all non-ground negative literals and we say the construction has floundered,

3.2) H_l is empty or

3.3) we can extend the path further.

proof:

Say that G is an n-path tree for H = $\leftarrow L_1, \cdots ,L_p$. We construct an n-path starting with the first goal G_0 = H Let α_0 take each L_l to the successor t_l of the root of the tree

llabelled L_l.

We keep constructing a path $(G_0, \cdots, G_l)$ with $(\theta_0, \cdots, \theta_{l-1})$ for as long as we can or want to. We shall ensure that it is always an example of **G**. The short path (G_0) is clearly an example.

Now suppose that we have a path of length l and we want to extend it. Obviously if G_l is empty then we can not. Also, if G_l is all non-ground negative literals then we must give up and in this case we say that the construction has floundered. Otherwise **R** will choose any positive or ground negative literal.

Suppose that G_l is $\leftarrow K_1, \cdots, K_r$ and that K_j is chosen. There are three cases.

Case 1 : $\alpha_l(K_j)$ is clabelled "unfinished". Again we must abandon our construction here. Note that this case will not arise if **G** is finished.

Case 2 : $\alpha_l(K_j)$ is llabelled positively, say with $P(\pi)$, and clabelled with a clause $P(\sigma) \leftarrow A_1, \cdots, A_q$. If the mgus of K_j from G_l and $P(\sigma)$ from the clause are θ_l , ϕ_l then G_{l+1} is $\leftarrow K_1\theta_l, \cdots, K_{j-1}\theta_l, A_1\phi_l, \cdots, A_q\phi_l, K_{j+1}\theta_l, \cdots, K_r\theta_l$. Since **G** is a path tree, by PT3 there is a substitution ζ of the variables in the clause such that $\sigma\zeta=\pi$. We put $\alpha_{l+1}(K_u\theta_l)=\alpha_l(K_u)$ and $\alpha_{l+1}(A_u\phi_l)$ equal to the successor s_u of $\alpha_l(K_j)$ with llabel $A_u\zeta$.

The extended path ($G_0, \cdots, G_{l+1}$) with ($\theta_0, \cdots, \theta_l$) is an example of **G** as we now show.

Since the path up to G_l is an example of **G**, there is, by EG1, a substitution ϵ_l such that $G = G_0\theta_0 \cdots \theta_{l-1}\epsilon_l$. By EG3, $\alpha_l(K_j)$ is llabelled $P(\pi) = K_j\epsilon_l$. So $K_j\epsilon_l=P(\sigma\zeta)$. But θ_l , ϕ_l are mgus of K_j and $P(\sigma)$ so there is ϵ_{l+1} such that $\epsilon_l=\theta_l\epsilon_{l+1}$ and $\zeta=\phi_l\epsilon_{l+1}$. Thus $G = G_0\theta_0 \cdots \theta_{l-1}\theta_l\epsilon_{l+1}$ and EG1 holds.

Since $K_j\epsilon_l=K_j\theta_l\epsilon_{l+1}$, this is what we require as the llabel of the image ($= \alpha_l(K_j)$) of

$K_j\theta_l$ under α_{l+1}. We also have $\alpha_{l+1}(A_u\phi_l)$ being a node labelled $A_u\zeta=A_u\phi_l\epsilon_{l+1}$ as required. Because $\epsilon_l=\theta_l\epsilon_{l+1}$ we have the other α_l behaving correctly in respect to the longer path as they did in respect to the path up to G_l. So EG2 holds.

K_j is chosen from G_l to get G_{l+1} and it is easy to see that EG3 holds.

Hence we have constructed our longer example.

Case 3: $\alpha_l(K_j)$ is llabelled with a ground negative literal, not(P (σ)) say, P(σ) finitely fails in QEP at level n-1, $\alpha_l(K_j)$ has no successors and is clabelled "negative". By EG2, not(P(σ)) = $K_j\epsilon_l$ but we only choose ground negative K_j so K_j is not(P(σ)). We have a new goal, $G_{l+1} = \leftarrow K_1, \cdots, K_{j-1}, K_{j+1}, \cdots, K_r$ with θ_l being the identity. If we get α_{l+1} to map as α_l on those literals and put ϵ_{l+1} equal to ϵ_l then it is straight forward to check that we still have an example.

Q.E.D. (lemma 4.5)

LEMMA 4.6

Suppose that G-path is a successful path for G under θ, that H-path is a path for H_0, that H_0 is a substitution of $G\theta$ and that both G-path and H-path are examples of a path tree G. Then H-path does not end full of non-ground negative literals.

proof:

Suppose that H-path is $(H_0, \cdots, H_k)$ with $(\psi_0, \cdots, \psi_{k-1})$ and suppose for contradiction that H-path does end in all non-ground negative literals. By lemma 4.4, we can construct an n-path tree H for $H_0\psi_0 \cdots \psi_{k-1}$ such that H-path is an example of H with ϵ being the identity and the range of α_k being the unfinished nodes of H.

By lemma 4.7, **H** will be a sub-tree of **G**.

By lemma 4.8, since $H_0\psi_0 \cdot \cdot \cdot \psi_{k-1}$ is a substitution of $G\theta$, we have an example, I-path, of **H** which is either successful or there is a literal K which is the one selected from a goal G_m in G-path and a node of **H** which is clabelled "unfinished" and llabelled with a substitution of K.

If I-path was successful then, since it is an example of **H**, **H** is finished (by lemma 4.9). This can not be as the range of the map α_k from the non-empty last goal of H-path to the nodes of **H** is the unfinished nodes of **H**.

So the following is true. From G_m which is $\leftarrow K_1, \cdot \cdot \cdot, K_r$ say, K_j is selected, and a node t of **H** is clabelled "unfinished" and llabelled with a substitution of K_j.

Since t is unfinished it is $\alpha_k(M_r)$ for one of the literals M_r from H_k. By EG2, the llabel of $\alpha_k(M_r)$ is $M_r\epsilon$ but ϵ is the identity. So the llabel of t is M_r itself. However, we know that this llabel is a substitution of K_j. Since M_r is non-ground negative so must K_j be.

This contradicts the fact that it was selected from G_m in the successful G-path.

Q.E.D. (lemma 4.6)

LEMMA 4.7

Suppose that **G** is an n-path tree, we construct an n-path, H-path, which is an example of **G**, as in construction two, and we construct, from H-path, an n-path tree H as in lemma 4.4 so that H-path is an example of **H**. Then **H** is a sub-tree of **G**.

proof:

Suppose that H-path is $(H_0, \cdots, H_k)$ with $(\psi_0, \cdots, \psi_{k-1})$. If G is a path tree for G $= \leftarrow L_1, \cdots, L_p$ then, since H-path is an example of G, there is a substitution ϵ such that $G = H_0\psi_0 \cdots \psi_{k-1}\epsilon$.

The construction in lemma 4.4 proceeds by the construction of a sequence of trees H_0 , ... , H_k. H-path is an example of G and of each of the H_i so we will denote each of the literal appearance to node maps as α_i^G , $\alpha_i^{H_0}$, ... , $\alpha_i^{H_k}$ to distinguish them.

We show by induction on i that

IC1) each H_i is a sub-tree of G.

We also show that the map β_i from the nodes of H_i to those of G satisfies

IC2) if H_i is $\leftarrow K_1, \cdots, K_r$ and $\alpha_i^{H_i}(K_j) = t$ then $\beta_i(t) = \alpha_i^G(K_j)$ and $\gamma = \epsilon$.

First i=0. We recall that H_0 has a root with p successors, $t_1, \cdots, t_p$ with each t_i llabelled $K_i\psi_0 \cdots \psi_{k-1}$. G is an n-path tree for G so its root has p successors, $s_1, \cdots, s_p$ say. Each s_i is llabelled $L_i\theta$. We define β_0 as follows. $\beta_0(root_{H_0}) = root_G$ and $\beta_0(t_i) = s_i$. Its easy to check ST1 to ST5. ST4 holds because all H_0's non-root nodes are clabelled "unfinished". ST5 holds with $\gamma = \epsilon$ - we know $K_i\psi_0 \cdots \psi_{k-1}\epsilon = L_i$ because of the relationship between H_0 and G.

Now let us assume that the conditions hold for i and we try and move on to i+1. Suppose that H_i is $\leftarrow K_1, \cdots, K_r$ and that K_j is chosen.

case 1 : $\alpha_i^G(K_j)$ is clabelled "unfinished". This can not happen because then we would be at the end of H-path at H_i.

case 2: K_j is positive, $\alpha_i^G(K_j) = t$ is llabelled positively, say by $P(\bar{x})$, t is clabelled $P(\bar{y}) \leftarrow A_1, \cdots, A_q$, $\bar{y}\zeta = \bar{x}$, the mgus of K_j from H_i and $P(\bar{y})$ from the clause are ψ_i , ϕ_i and H_{i+1} is $\leftarrow K_1\psi_i, \cdots, K_{j-1}\psi_i, A_1\phi_i, \cdots, A_q\phi_i, K_{j+1}\psi_i, \cdots, K_r\psi_i$. So H_{i+1} has q new nodes - $s_1, \cdots, s_q$ the successors of $\alpha_i^{H_i}(K_j) = s$. We let β_{i+1} map as β_i on H_i and $\beta_{i+1}(s_u)$ is

the successor t_u of t which is llabelled $A_u\zeta$. This clearly makes IC2 hold.

ST1 still holds.

If t' has exactly p' >0 successors in $\mathbf{H}_{l+1}$ then either

a) it did so in $\mathbf{H}_l$ so β_l mapped them one-to-one to the exactly p' successors of $\beta_l(t')$ in G so so does β_{l+1} or

b) t'=t and p'=p whence β_{l+1} maps $s_1, \cdots, s_q$ to $t_1, \cdots, t_q$ respectively in G.

So ST2 holds.

If $\beta_{l+1}(t')$ has a successor $\beta_{l+1}(u')$ in G then either

a) it did so in $\mathbf{H}_l$ so u' is a successor of t' in $\mathbf{H}_l$ so in $\mathbf{H}_{l+1}$ or

b) u' is one of the s_u, t' is s whence u' is a successor of t'.

So ST3 holds.

The only new clabel to check is that of s which is also that of t so ST4 holds.

We know that $K_j\psi_l=P(\sigma\phi_l)$, that s is llabelled $K_j\psi_l\psi_{l+1}\cdots\psi_{k-1}$ (as H-path is an example of $\mathbf{H}_l$) so t is llabelled $K_j\psi_l\cdots\psi_{k-1}\epsilon$. However, this is $P(\pi)$ so $P(\pi) = K_j\psi_l\cdots\psi_{k-1}\epsilon = P(\sigma)\ \phi_l\psi_{l+1}\cdots\psi_{k-1}$. Thus $\zeta = \phi_l\psi_{l+1}\cdots\psi_{k-1}\epsilon$. Now s_u is llabelled $A_u\phi_l\psi_{l+1}\cdots\psi_{k-1}$ and $\beta_{l+1}(s_u) = t$ is llabelled $A_u\zeta=A_u\phi_l\psi_{l+1}\cdots\psi_{k-1}\epsilon$. So ST5 holds.

Case 3 : K_j is negative. This is almost just a simple version of case 2.

Q.E.D. (lemma 4.7)

LEMMA 4.8

If G-path, $(G_0, \cdots, G_l)$ with $(\theta_0, \cdots, \theta_{l-1})$, is a successful n-path for $G = G_0$, $\theta = \theta_0\cdots\theta_{l-1}$ is an example of an n-path tree **G** , **H** is a path tree for H which is a substitution of $G\theta$ and **H** is a sub-tree of **G** then we can build an n-path I-path which is an

example of H and is either successful or there is G_m and a node of H such that if K is selected from G_m then the llabel of t is a substitution of K and the clabel of t is "unfinished".

proof:

We use lemma 4.5 to construct I-path $=(I_0, \cdots, I_k)$ with $(\zeta_0, \cdots, \zeta_{k-1})$ from H but selecting literals so that

1) for each i there is η_i such that $G_i\eta_i=I_i$,

2) and either

2.1) k=l (and so the I-path is successful) or

2.2) literal K is selected from G_k but $\alpha_k^{I\to H}(K\eta_k)$ which has llabel a substitution of K is clabelled "unfinished".

By induction on i, we find η_i such that

IC1) $G_i\eta_i=I_i$ and

IC2) if α_i^H is the map mapping the literals in I_i to nodes in H, β is the sub-tree mapping nodes in H to nodes in G and α_i^G is the map mapping literals in G_i to nodes in G then for all K_j in G_i, $\beta(\alpha_i^H(K_j\eta_j))=\alpha_i^G(K_j)$.

$H_0=G_0\theta_0 \cdots \theta_{i-1}=G\theta$. H is a tree for $H_0\psi_0 \cdots \psi_{k-1}=G\theta\psi$ so $I_0=G\theta\psi=G_0\theta\psi$. $\beta(s_i)=t_i$. So the conditions hold when i=0.

Suppose that we have constructed $(I_0, \cdots, I_i)$ with $(\zeta_0, \cdots, \zeta_{i-1})$ from H such that IC1 and IC2 hold.

If i=l we are done, so suppose that i < l. Then G_i is non-empty so so is I_i which equal $G_i\eta_i$. Since the G-path does not flounder G_i is not all non-ground negative literals so neither is I_i. Say G_i is $\leftarrow K_1, \cdots, K_r$ and K_j is selected to get to G_{i+1}. We select $K_j\eta_i$ from I_i to continue I-path. Since K_j is not non-ground negative, neither is $K_j\eta_i$. There

are three cases.

Case 1 : $\alpha_i^H(K_j\eta_i)$ is clabelled unfinished. Then we must stop our construction here. We put i=k and we have our result.

Case 2 : $\alpha_i(K_j\eta_i)$ is llabelled positively, say with P(π), and clabelled with a clause, say P(ν) $\leftarrow A_1, \cdots, A_q$. I_{i+1} is got from I_i by unifying $K_j\eta_i$ with the head of the clause. Now $\alpha_i(K_j\eta_i)$ is a node in H which is a sub-tree of G but $\beta(\alpha_i^H(K_j\eta_i)) = \alpha_i^G(K_j)$ so $\alpha_i^G(K_j)$ is clabelled with the clause (by ST4). Since K_j is selected to get to G_{i+1}, and G-path is an example of G then EG3 implies that if θ_i , ϕ_i are the mgus of K_j from G_i and P(ν) from the clause then G_{i+1} is $\leftarrow K_1\theta_i, \cdots, K_{j-1}\theta_i, A_1\phi_i, \cdots, A_q\phi_i, K_{j+1}\theta_i, \cdots, K_r\theta_i$. If ζ_i , ξ_i are the mgus of $K_j\eta_i$ from I_i and P(ν) from the clause then I_{i+1} is $\leftarrow K_1\eta_i\zeta_i, \cdots, K_{j-1}\eta_i\zeta_i, A_1\xi_i, \cdots, A_q\xi_i, K_{j+1}\eta_i\xi_i, \cdots, K_r\eta_i\zeta_i$. But $K_j\eta_i\zeta_i = P(\nu)\xi_i$ so there is an η_{i+1} such that $\eta_i\zeta_i = \theta_i\eta_{i+1}$ and $\xi_i = \phi_i\eta_{i+1}$. Thus $I_{i+1} = G_{i+1}\eta_{i+1}$.

By ST2, β maps the successors of $\alpha_i^H(K_j\eta_i)$ to the successors of $\alpha_i^G(K_j)$ in such a way that IC2 is preserved.

Case 3 : $\alpha_i(K_j\eta_i)$ is llabelled with a negative ground literal and clabelled "negative". This case is straight forward in the light of case 2.

Q.E.D. (lemma 4.8)

LEMMA 4.9

If $(G_0, \cdots, G_l)$ with $(\theta_0, \cdots, \theta_{l-1})$ is a successful path and is an example of G then G is finished.

proof:

Suppose that G is unfinished. By EG3, any node of G in the image of any α_i is clabelled with a clause or "negative". So any of the unfinished nodes are not in the range of any of the α_i. Let t be a minimal such node - i.e. t is not in the range of any of these maps but its predecessor, s say, is. Suppose that s is $\alpha_i(K_j)$.

We follow K_j to $K_j\theta_i$ in G_{i+1}, to $K_j\theta_i\theta_{i+1}$ in G_{i+2}, ... and so on until $K_j\theta_i \cdots \theta_{i+p}$ is selected from G_{i+p+1}. This must eventually happen so that G_i can be empty.

By EG3, each $\alpha_{i+r}(K_j\theta_i \cdots \theta_{i+r-1})$ is equal to s. If $K_j\theta_i \cdots \theta_{i+p}$ was negative then, by EG3, s would have no successors. It does, so it is positive and the clabel of s is a clause with q body literals say. By EG3, s has exactly q successors and all are mapped to by α_{i+1}. This contradicts our assumptions about t.

Q.E.D. (lemma 4.9)

PART b) n is the minimum integer such that G finitely fails in the QEP at level n.

By induction on l we show that

if H is a goal , n is the minimum integer such that H finitely fails at level n in QEP and l is the minimum integer such that H has a closed bounded set of unsuccessful n-paths of length less than or equal to l
then H finitely fails at level n in R-qep.

First suppose that l=0. Say that G is the chosen set of closed bounded unsuccessful n-paths for G of length 0 which was used in the definition of **R**. Since **R** can not select a literal which extends the path (H) into G it must select a literal which immediately fails H. So we have a closed bounded set of unsuccessful **R** n-paths for H as required.

Now assume we are up to an l greater than zero. One set of closed bounded unsuccessful n-paths of length less than or equal to 1 for H are chosen in the definition of

R. Suppose that it is G. R chooses a literal from H which either immediately fails or extends (H) into a path in G. There are two cases depending on whether the literal is positive or negative but the proofs are similar so that we will only present the positive case.

Suppose that L is the chosen positive literal. For each clause which has a head unifying with L if H_1 is the result of unifying L from H with the clause and θ the appropriate mgu then (H , H_1) with (θ) is in G. If we take all the paths from G with second goal H_1 and chop of the first goal H then its easy to see that we end up with a closed bounded set of unsuccessful n-paths for H_1 all of length less than or equal to l-1. By our inductive hypothesis H_1 must finitely fail at level n in R-qep. If we now add H to the start of all the R-paths in a closed bounded set of unsuccessful R n-paths for H_1 then we know that H finitely fails at level n in R-qep.

Q.E.D. (of THEOREM 4.3)

Chapter 5

Declarative Meaning

It would make our job of relating LP with NAF to other query answering approaches much easier if we could characterise the answers to a program in some simpler way than as those queries successful in the complicated QEP. For similar reasons, other researchers, from Clark [Cla78] onwards, have tried to give LP with NAF a declarative meaning. We have already discussed what would be acceptable as such a meaning and why existing attempts fail. In this chapter we solve our problems by giving LP with NAF a satisfactory declarative meaning in the case when no function symbols appear.

5.1 The Synthesis of a Program

We now try to capture the meaning of negation as failure by describing a theory, SYNTH(PROG), corresponding to any logic program using NAF. The idea will be that making deductions from SYNTH(PROG) will give the same answers as running the program.

It must be noted that a successful attempt to capture the meaning of NAF must deal with the possible appearance of function symbols in programs. These are useful in many applications of logic programming. We suspect that the SYNTH alternative can be generalised in a straight forward way to programs containing function symbols but for our purposes we need only look at function-free programs.

We suppose that PROG is a LOGIC PROGRAM with NAF containing no function symbols. **L** is a finite relational signature containing all the relation symbols of PROG plus any others that one might want to query of PROG. **C** is a non-empty but finite set of constant symbols containing all those that appear in PROG plus any others that one might ask about in queries. The clauses in PROG are $c_1, \cdots, c_p$.

For each signature **L**, we suppose that we have determined disjoint signatures $\mathbf{L}^*$, L^*_{PROG}, $fail_{PROG}$ such that:

1) L^* contains exactly one n-ary symbol R^* for each n-ary R from **L**;

2) $\mathbf{L}^*_{PROG}$ contains exactly one n-ary symbol R^*_c for each clause c from PROG where c has n-ary R from **L** in its head;

3) $fail_{PROG}$ contains exactly one k-ary relation symbol $fail_c$ for each clause c from PROG, where c contains k different variable symbols.

We define $\mathbf{L}^{PROG}$ to be the union of these three signatures.

In all models of the theory SYNTH(PROG), R^* will be true of all the tuples which fail finitely to be proven to be in R from PROG. We intend R^*_c to be true of all those tuples which fail finitely to be proven to be in R from PROG using a resolution proof with top clause c. We intend $fail_c$ to be true of a tuple $\bar{a}$ if when the body of c is taken as a query with all the variables of c in some fixed order replaced by $\bar{a}$ then the query fails finitely.

The synthesis SYNTH(PROG) of PROG is the finite first-order $\mathbf{L} \cup \mathbf{L}^{PROG} \cup \mathbf{C}$-theory defined by

SYNTH(PROG) = POS(PROG) ∪ NEG(PROG) ∪ INEQ(PROG).

In fact, it is dependent on the choice of **L** and **C**. When we want to be specific it will be introduced as SYNTH(PROG , **L** , **C**).

POS(PROG)={ POS(c_i) | i=1...p }.

NEG(PROG)={ NEG(R) | $R \in$ **L** } ∪ NEG(c_1) ∪ ··· ∪ NEG(c_p).

INEQ(PROG)= { a ≠ b | a,b different constants from **C** }.

For every k-ary relation symbol from **L** either

1) R does not appear in the heads of any clauses in PROG, in which case NEG(R) is the sentence $\forall x\ R^*(x)$; or

2) R appears in the heads of precisely the clauses $d_1, \ldots, d_q$, in which case NEG(R) is the sentence $\forall x\ [\ \wedge_{j=1}^{q} R^*_{d_j}(x) \rightarrow R^*(x)\ ]$.

For each clause c, we define the sentence POS(c) and the theory NEG(c) as follows. Suppose that c is $P(v) \leftarrow A_1, \ldots, A_n$, n is a non-negative integer, the distinct variables appearing in c are $x=(x_1, \ldots, x_m)$ in some order, P is k-ary, and $v=(v_1, \ldots, v_k)$.

In what follows, for a literal A, if A is positive, say $R(a)$, we put $A^+ = R(a)$ and $A^- = R^*(a)$ and if A is negative , say not($R(a)$), we put $A^+ = R^*(a)$ and $A^- = R(a)$.

POS(c) for c as above is $\forall x\ (\ \wedge_{i=1}^{n} A_i^+ \rightarrow P(v)\ )$.

Suppose that $w=(w_1, \ldots, w_l)$ are precisely the distinct variables appearing in v, that $a=(u_1, \ldots, u_k)$ are k new variable symbols, and that $z=(z_1, \ldots, z_r)$ are the distinct variables from x not appearing in v. Then NEG(c)= NEGV(c) ∪ NEGC(c) ∪ NEGB(c) ∪ FAIL(c) where

NEGV(c) =

{ $\forall a\ (\ u_i \neq u_j \rightarrow P^*_c(a)\ )$ | i≠j are from 1...k such that v_i and v_j are the same variable }

NEGC(c) = { $\forall a\ (\ u_i \neq$ a $\rightarrow P^*_c(a)\ )$ | i=1...k v_i=a from **C** }

NEGB(c) = { $\forall w\ (\ (\ \forall z\ fail_c(x)\ ) \rightarrow P^*_c(v)\ )$ }

FAIL(c) = { $\forall x\ (\ A_i^- \rightarrow fail_c(x)\ )$ | i=1...n }.

For example if PROG is

P(a) ← not(P(a))

R(x) ← Q(x,y) , T(y) , not(S(x))

Q(x,x) ←

T(a) ←

S(b) ←

from a set of constants {a,b} and a signature including only the relation symbols in PROG then SYNTH(PROG) is

{ $P^*(a) \rightarrow P(a)$,

$\forall xy(Q(x,y) \wedge T(y) \wedge S^*(x) \rightarrow R(x))$,

$\forall x$ (Q(x,x)) ,

T(a) ,

S(b) ,

$\forall x(P_1^*(x) \rightarrow P^*(x))$, $\forall xy(Q_3^*(x,y) \rightarrow Q^*(x,y))$,

$\forall x(R_2^*(x) \rightarrow R^*(x))$, $\forall x(S_5^*(x) \rightarrow S^*(x))$,

$\forall x(T_4^*(x) \rightarrow T^*(x))$,

$\forall u$ ($u \neq a \rightarrow P_1^*(u)$) , $fail_1 \rightarrow P_1^*(a)$, $P(a) \rightarrow fail_1$,

$\forall x$ (($\forall y$ $fail_2(x,y)$) $\rightarrow R_2^*(x)$) , $\forall xy$ ($Q^*(x,y) \rightarrow fail_2(x,y)$) ,

$\forall xy$ ($T^*(y) \rightarrow fail_2(x,y)$) , $\forall xy$ ($S(x) \rightarrow fail_2(x,y)$) ,

$\forall uv$ ($u \neq v \rightarrow Q_3^*(u,v)$) , $\forall x(fail_3(x) \rightarrow Q_3^*(x,x))$,

$\forall u$ ($u \neq a \rightarrow T_4^*(u)$) , $fail_4 \rightarrow T_4^*(a)$,

$\forall u$ ($u \neq b \rightarrow S_5^*(u)$) , $fail_5 \rightarrow S_5^*(a)$,

$a \neq b$ }.

It is interesting to note that $S^*(a)$, $T^*(b)$, $R^*(b)$, R(a) follow from SYNTH(PROG) but neither P(a) nor $P^*(a)$ does while our query evaluation process on PROG finitely fails S(a), T(b) and R(b), succeeds R(a) and does not terminate on P(a).

5.2 Soundness of the QEP with respect to SYNTH

In this section we give a detailed outline of the proof that the negation as failure query evaluation process is "sound" with respect to the synthesis of a program. This means that the answers the QEP comes up with are true in all models of the synthesis of the program.

To be precise, we only prove soundness for atomic propositions. This is not a serious limitation because, if we want to know about answers to $L_1, \cdots, L_p$ with variables x then we could choose a new symbol Q put $Q(x) \leftarrow L_1, \cdots, L_p$ in the program and ask "$Q(x)$?".

As usual, if α is a formula with free variables x then $\forall\alpha$ is the sentence $\forall x \alpha$.

THEOREM 5.1

For all signatures **L**, for all finite sets **C** of constant symbols, for all logic programs PROG with symbols from **L**∪**C**, for all non-negative integers r, for all r-ary R from L, for all r-tuples a of variable symbols and constants, for all substitutions θ of the variables in a

if R(a) succeeds with answer θ

then SYNTH(PROG) $\vdash \forall R(a\theta)$.

proof:

First some notation for the proof. If L is the positive literal R(a) then L^+ is L itself while L^- is $R^*(a)$. If L is the negative literal not(R(a)) then L^+ is $R^*(a)$ while L^- is R(a). If B is the list $L_1, \cdots, L_p$ of literals then B^+ is the conjunction $L_1^+ \wedge \cdots \wedge L_p^+$ while B^- is the disjunction $L_1^- \vee \cdots \vee L_p^-$.

We actually show that for all n,

a) if R($\bar{x}$) succeeds with answer θ by level n then SYNTH(PROG) $\vdash \forall R(\bar{x}\theta)$, and

b) if R($\bar{x}$) finitely fails by level n then SYNTH(PROG) $\vdash \forall R'(\bar{x})$.

To do so we proceed by induction on n. Suppose a) and b) are true for all m less than n. Now let us look at n.

a) R($\bar{x}$) succeeds with answer θ by level n

We show that $SYNTH(PROG) \vdash \forall R(\bar{x}\theta)$ by taking any model M of $SYNTH(PROG)$ and showing that $M \models \forall R(\bar{x}\theta)$. We must point out that M does not have to be finite.

We know that there is an n-path

$(\leftarrow B_0, \leftarrow B_1, \cdots, \leftarrow B_k, \leftarrow)$

such that B_0 is R($\bar{x}$), each $\leftarrow B_{i+1}$ is got from $\leftarrow B_i$ using a unifier θ_i and $\theta = \theta_0 \cdots \theta_k$.

We proceed by induction back from B_k to B_0 showing that for each i,

$M \models \forall(B_i^* \theta_i \cdots \theta_k)$.

Then $M \models \forall R(\bar{x}\theta)$ as required. So suppose that either $i=k$ or that $i<k$ and

$M \models \forall(B_{i+1}^* \theta_{i+1} \cdots \theta_k)$.

We must show that $M \models \forall(B_i^* \theta_i \theta_{i+1} \cdots \theta_k)$.

There are two cases:

1) the path is continued by the selection of a positive literal from $\leftarrow B_i$; or

2) the path is continued by the selection of a negative literal.

case 1

B_i is $L_1, \cdots, L_p$, the path is continued by the selection of a positive literal L_j from $L_1, \cdots, L_p$, and L_j unifies with the head A of the clause $A \leftarrow A_1, \cdots, A_q$ in PROG. If the mgus of L_j from $\leftarrow B_i$ and A from the clause are θ_i , ϕ then B_{i+1} is

$L_1\theta_l, \cdots, L_{j-1}\theta_l, A_1\phi, \cdots, A_q\phi, L_{j+1}\theta_l, \cdots, L_p\theta_l$.

Thus we know that

$M \models \forall(L_1^*\theta_l \wedge \cdots \wedge L_{j-1}^*\theta_l \wedge A_1^*\phi \wedge \cdots \wedge A_q^*\phi \wedge L_{j+1}^*\theta_l \wedge \cdots \wedge L_p^*\theta_l)\theta_{l+1} \cdots \theta_k$

or that $l=k$, $B_k = L_j$ and the clause is just $A \leftarrow$ in the program.

It is not hard to use the sentence $\forall x(A_1^* \wedge \cdots \wedge A_q^* \rightarrow A)$ in $SYNTH(PROG)$ and the fact that $L_j^*\theta_l = L_j\theta_l = A\phi$ to show that $M \models \forall(B_l^*\theta_l)\theta_{l+1} \cdots \theta_k$ as required.

case 2

B_l is $L_1, \cdots, L_p$, L_j = not(P(x)) is selected from $L_1, \cdots, L_p$ to continue the path, θ_l is the identity substitution on the variables in x, P(x) finitely fails at level n-1 and B_{l+1} is $(L_1, \cdots, L_{j-1}, L_{j+1}, \cdots, L_p)\theta_l$.

By our inductive hypothesis $SYNTH(PROG) \vdash \forall(P^*(x\theta_l))$ so $M \models \forall(P^*(x\theta_l))$ and $M \models \forall(L_l^*\theta_l)$. Its easy to deduce that $M \models \forall(L_l^*\theta_l\theta_{l+1} \cdots \theta_k)$.

It is a simple matter to combine this result with the fact that either B_{l+1} is empty or that $M \models \forall(B_{l+1}^*\theta_l\theta_{l+1} \cdots \theta_k)$ (as θ_l is the identity) to get that

$M \models \forall(L_1^* \wedge \cdots \wedge L_{j-1}^* \wedge L_j^* \wedge L_{j+1}^* \wedge \cdots \wedge L_p^*)\theta_l \cdots \theta_k$

as required.

b) R(x) finitely fails by level n

We show that $SYNTH(PROG) \vdash \forall R^*(x)$ by again taking a model M of $SYNTH(PROG)$ and showing that $M \models \forall R^*(x)$.

We know that there is a closed bounded set G of unsuccessful n-paths for R(x). Say that all of its paths have length less than l. We proceed by decreasing induction on k down from l to 1 showing that if a path from G of length k ends in $\leftarrow B$ then $M \models \forall B^-$.

Since $(\leftarrow R(\vec{x}))$ is itself in G this we will show that $M \models \forall R^*(\vec{x})$.

For k=l there is nothing to prove since there are no paths in G of length l. So suppose that we have proven the inductive hypothesis for paths of length k+1 and that $(\leftarrow R(\vec{x}), S_1, \cdots, S_{k-1})$ is of length k from G.

S_{k-1} can not be the empty goal so suppose that it is $\leftarrow L_1, \cdots, L_p$. Because G is closed we know that there is an L_i such that either:

case 1 : L_i is positive and if it unifies with the head of a program clause $A \leftarrow A_1, \cdots, A_q$ with mgus θ, ϕ of L_i from S_{k-1} and A from the clause and S_{k+1} is $\leftarrow L_1\theta, \cdots, L_{i-1}\theta, A_1\phi, \cdots, A_q\phi, L_{i+1}\theta, \cdots, L_p\theta$ then $(\leftarrow R(\vec{x}), S_1, \cdots, S_k, S_{k+1})$ is in G; or

case 2 : L_i is not(P($\vec{\tau}$)) which is ground, n is greater than zero and P($\vec{\tau}$) succeeds at level n-1 ; or

case 3 : L_i is not(P($\vec{\tau}$)) which is ground, n is greater than zero, P($\vec{\tau}$) finitely fails at level n-1 and $(\leftarrow R(\vec{x}), S_1, \cdots, S_k, \leftarrow L_1, \cdots, L_{i-1}, L_{i+1}, \cdots, L_p)$ is in G.

In case 2, our inductive hypothesis (on n) tells us that $M \models \forall L_i^-$ from which we have our desired result that $M \models \forall(L_1^- \vee \cdots \vee L_p^-)$.

In case 3, our inductive hypothesis on l tells us that

$M \models \forall(L_1^- \vee \cdots \vee L_{i-1}^- \vee L_{i+1}^- \vee \cdots \vee L_p^-)$

which also implies our desired result.

In case 1, if there are no clauses in PROG containing the relation symbol P , say, which is in L_i then $SYNTH(PROG)$ contains $\forall \vec{x} P^*(\vec{x})$ so that $M \models \forall L_i^-$ from which we get our result.

The possibility remains that in case 1 there are some clauses in PROG with P in their heads. Suppose they are $d_1, \cdots, d_g$.

We know that $SYNTH(PROG)$ contains the sentence $\forall x(\wedge_{i=1}^{g} P^*_{d_i}(x) \rightarrow P^*(x))$. Thus, it is sufficient for showing our result to show that for each substitution ϕ of the variables in $L_1, \cdots, L_p$ by elements of the domain of M either $M \models (L_1 \vee \cdots \vee L_{i-1} \vee L_{i+1} \vee \cdots \vee L_p)\phi$ or for each clause d from $d_1, \cdots, d_g$, $M \models P^*_d(x\phi)$.

Given any substitution ϕ suppose that the first possibility does not hold and that d is one of the d_i. Say d is $P(v) \leftarrow A_1, \cdots, A_q$.

There are two cases. First suppose that $x\phi$ can not be got as a substitution of v. By TR12 we have either:

1) there is an i such that v_i is a constant a and $u_i\phi$ is not the interpretation of a in M; or

2) there are i and j such that $v_i = v_j$ is a variable symbol but $u_i\phi$ and $u_j\phi$ are different elements of M.

Sentences in NEGV(d) and NEGC(d) give the result that $M \models P^*_d(x\phi)$ as required.

Next we consider the case when $x\phi$ is equal to $v\eta$ for some M-substitution η of the variables w in v.

We use the sentence $\forall w(\forall z\, fail_d(x) \rightarrow P^*_d(v))$ which M satisfies (because it is in $SYNTH(PROG)$) to prove that $M \models P^*_d(v\eta)$. Note that x are all the variables in d and z are those x_i not in w. So given any M-substitution μ of z we must show that $M \models fail_d(x)(w \backslash w\eta, z \backslash z\mu)$.

Remembering that θ, χ are the mgus of $P(x)$ and $P(v)$ we know by our induction on k and the fact that G is closed that

$M \models \forall(L_1\theta \vee \cdots \vee L_{i-1}\theta \vee A_1\chi \vee \cdots \vee A_q\chi \vee L_{i+1}\theta \vee \cdots \vee L_p\theta)$.

For the same reasons plus the fact that $P(x\phi) = P(v)(w \backslash w\eta, z \backslash z\mu)$, we know that there is a substitution ϵ of the new goal such that $\theta\epsilon = \phi$ and $\chi\epsilon = (w \backslash w\eta, z \backslash z\mu)$. Then it is clear that each $L_r\theta\epsilon = L_r\phi$ and each $A_s\chi\epsilon = A_s(w \backslash w\eta, z \backslash z\mu)$. Since

$M \models (L_1\theta \vee \cdots \vee L_{i-1}\theta \vee A_1\chi \vee \cdots \vee A_q\chi \vee L_{i+1}\theta \vee \cdots \vee L_p\theta)\epsilon$,

we have

$M \models L_1\phi \vee \cdots \vee L_{i-1}\phi \vee$

$A_1(w\backslash w\eta, z\backslash z\mu) \vee \cdots \vee A_q(w\backslash w\eta, z\backslash z\mu) \vee$

$L_{i+1}\phi \vee \cdots \vee L_p\phi.$

Because we assumed that M is a model of no $L_j\phi$ we know that $M \models A_r(w\backslash w\eta, z\backslash z\mu)$ for some r. However, we also know that $M \models \forall x(A_r \rightarrow fall_d(x))$. Hence $M \models fall_d(x)(w\backslash w\eta, z\backslash z\mu)$ as required.

Q.E.D.

5.3 Correctness of the QEP with respect to SYNTH

In this section we will see how correctly the QEP captures consequences of SYNTH(PROG). We will establish that the QEP never says "no" to atomic consequences of SYNTH(PROG).

THEOREM 5.2

For all signatures **L**, for all non-empty finite sets **C** of constant symbols, for all logic programs PROG with symbols from **L**∪**C**, for all non-negative integers r, for all r-ary R from **L**, for all r-tuples $\bar{a}$ of constants from **C**

if SYNTH(PROG) $\vdash R(\bar{a})$

then R($\bar{a}$) succeeds or fails to terminate in the QEP.

proof:

The basic idea of the proof is to construct a model of SYNTH(PROG) which contains information about the operation of the QEP. We construct an $\mathbf{L} \cup \mathbf{L}^{PROG} \cup \mathbf{C}$-structure M

as follows:

1) the domain of M is the set of constants C;

2) each constant is interpreted as itself;

3) for each S∈L, for each tuple $\bar{b}$ from C, $M \models S(\bar{b})$ if and only if the QEP either succeeds or does not terminate on $S(\bar{b})$;

4) for each S∈L, for each tuple $\bar{b}$ from C, $M \models S^*(\bar{b})$ if and only if the QEP either finitely fails or does not terminate on $S(\bar{b})$;

5) for each S∈L, for each clause d with S in its head, if the head of d is $S(\bar{v})$, its body is $A_1, \cdots, A_q$, $\bar{w}$ are the distinct variables in $\bar{v}$ in the order used in SYNTH(PROG), $\bar{z}$ are the others in d also in order and $\bar{x}$ are all the distinct variables in d in order then

5a) for each tuple $\bar{b}$ from C, $M \models S_d^*(\bar{b})$ if and only if

5a.1) $v_i = v_j$ is a variable symbol but $b_i \neq b_j$;

5a.2) v_i is the constant a from C but $b_i \neq a$; or

5a.3) there is a ground substitution ϕ for $\bar{w}$ such that

5a.3a) $\bar{v}\phi = \bar{b}$ and

5a.3b) for all ground substitutions ψ for $\bar{z}$ there is an i such that $M \models A_i(\bar{w}\backslash\bar{w}\phi, \bar{z}\backslash\bar{z}\psi)$; and

5b) for each tuple $\bar{a}$ from C $M \models fail_d(\bar{a})$ if and only if there is an i such that $M \models A_i(\bar{x}\backslash\bar{a})$.

Now we show that M is a model of SYNTH(PROG). We will prove the two lemmas later.

1) For all clauses c, $M \models POS(c)$ by lemma 5.4.

2) For all R from L, $M \models NEG(R)$ because either

2a) R appears in no heads,when for all $\bar{b}$ $R(\bar{b})$ finitely fails so $M \models R^*(\bar{b})$ and hence

$M \models \forall x R'(x)$; or

2b) R appears in precisely the heads of $d_1, \cdots, d_g$, when lemma 5.5 implies $M \models \forall x (\wedge_{i=1}^{g} R^*_{d_i}(x) \rightarrow R'(x))$ as required.

3) For all clauses c, by the definitions of the interpretations of $fall_c$ and S^*_c, $M \models NEG(c)$.

4) By the construction of M, $M \models INEQ(PROG)$.

Since SYNTH(PROG) $\vdash R(a)$, we know $M \models R(a)$ and by definition $R(a)$ either succeeds or does not terminate. Thus it only remains for us to prove the two crucial lemmas.

In order to prove lemma 5.4, we shall make use of some game theory in the tradition of Ehrenfeucht [Ehr61] and Fraisse [Fra54]. In particular, we shall introduce some games for two players such that the existence of a winning strategy for one particular player is equivalent to the finite failure of a given goal in the QEP operating on our program PROG.

The two players are called respectively the existential player and the universal player. The game proceeds as the development of a sequence of current goals. The n-game for $P(v)$ is played according to the following rules:

G0) the current goal starts as $\leftarrow P(v)$;

G1) if the current goal is empty the universal player wins;

G2) otherwise the existential player chooses a literal L from the current goal;

G3) if L is positive and does not unify with the head of any program clause then the existential player wins;

G4) if L is positive and does unify then the universal player picks one such clause which we use to get a new goal from the current goal in the usual way and this becomes the new current goal;

G5) if L is not($S(a)$) which is ground and $S(a)$ succeeds at level n-1 with any

substitution then the existential player wins;

G6) if L is not($S(\sigma)$) which is ground and $S(\sigma)$ finitely fails at level n-1 then the current goal with L left out becomes the new current goal;

G7) if L is not($S(\sigma)$) and $S(\sigma)$ is not ground or neither succeeds nor finitely fails at level n-1 then the universal player wins;

G8) if the game goes on forever then the universal player wins.

LEMMA 5.3

The existential player has a winning strategy for the n-game for $P(\tau)$ if and only if $P(\tau)$ finitely fails at level n.

proof:

the only if direction (essentially Konig's lemma)

Suppose that the existential player whose name is Eloise has a winning strategy. Let G be the set of all initial segments of the paths of the current goals got in all possible plays of the n-game for $P(\tau)$ with Eloise using her winning strategy. It is easy to see that G is closed and does not contain any successful paths.

We must show that G is bounded i.e. that there is a bound on the length of games. We know that no games go on forever because then Eloise would lose but that is a different matter to there being a bound on all their lengths.

Suppose for contradiction that there is no bound. We say that a play of the game is *possible*, now, if it is legal and Eloise plays according to her winning strategy. We show how Abelard can play so that at any stage of the play and for any integer m, there is a possible play of the game the same as the actual play up to that stage but going on for longer than stage m. Because there is no bound on the lengths of possible plays this is true

at the beginning.

Now suppose that it holds at some stage. The goal can not be empty for then Abelard would win. Eloise now picks a literal according to her strategy. Because the unbounded set of possible extensions of the actual play (as it was before Eloise made her move) consisted wholly of plays continued by Eloise moving according to her winning strategy, they are all still extensions of the actual game (after Eloise has made her move). If Abelard is not required to make a choice then we are at the next stage of the game with the condition still holding.

If, on the other hand, Abelard is required to make a choice we can get him to choose as follows so that the condition is still preserved. Abelard is only active when he is picking a clause from the program after Eloise has chosen a positive literal. Then he only has a finite number of choices. Suppose that each of these choices led to a stage which could only be extended by at most some finite amount of possible play. Then, contrary to what we have established, the play after Eloise's choice can only be extended by at most the finite maximum of these finite number of finite bounds. So there is a choice open to Abelard which leaves the condition holding. We get Abelard to make that choice.

Now, if Abelard plays according to that strategy the game will go on forever and he will win. Since Eloise is using her winning strategy we have our desired contradiction.

the if direction

Suppose that $P(\bar{x})$ finitely fails at level n. Let G be a closed bounded set of unsuccessful n-paths for $\leftarrow P(\bar{x})$. We show that Eloise has a winning strategy in the n-game for $P(\bar{x})$. Eloise will keep a record of the progress of play in a path of the current goals which we will show is always in G.

At the start of the game the first current goal is $\leftarrow P(\bar{x})$ and Eloise puts that in her

path. This short path is in G.

At each move of the play the record is kept in G as follows:

1) Since the current record is in G the current goal is not the empty one.

2) Also, because the current record is in the closed set G, there is a literal L in the non-empty current goal such that

2a) L is positive, and if it unifies with a head of a clause then there is an extension of the record using L and the clause in G,

2b) L is ground negative with the positive part of it succeeding at level n-1 (which is greater than or equal to zero).

or 2c) L is negative, ground, finitely failing at level n-1 and the extension of the record with L disappearing is in G.

We get Eloise to choose one such L.

3) If L is positive and does not unify then Eloise wins.

4) If L is positive and does unify then no matter which such clause Abelard chooses the choice of L guarantees that our next current goal keeps the record in G.

5) If L is not(P(x)) which is ground with P(x) succeeding at level n-1 then Eloise wins.

6) If L is not(P(x)) with P(x) ground and finitely failing at level n-1 then the choice of L guarantees that the next current goal keeps the record in G.

7) The choice of L guarantees that 3 to 6 are the only possibilities.

8) The game can not go on forever as the record is kept in G which is bounded.

There is no way that Abelard can win and as the game finishes eventually Eloise must win.

Q.E.D. (lemma 5.3)

Now let us prove the first of our two main lemmas.

LEMMA 5.4

If $A \leftarrow A_1, \cdots, A_q$ is a clause in PROG then $M \models \forall x(\wedge_{i=1}^{q} A_i^* \rightarrow A)$.

proof:

Suppose that ϕ is a substitution of x by elements from M (i.e. constants from C) such that for each i=1...q $M \models A_i^* \phi$.

We must prove that $M \models A\phi$. Suppose for contradiction that $A\phi$ finitely fails , say at level n. We are using the result that finite failure and success are incompatible. Thus the existential player has a winning strategy in the n-game for $A\phi$. We get a contradiction by inviting Eloise to play as the existential player in the n-game for $A\phi$. She can use the winning strategy yet we will show how we can win as the universal player.

To help us play we enlist the support of two people $player_i$ and $opponent_i$ for each i=1...q such that A_i is a positive literal. We get $player_i$ and $opponent_i$ to play the n-game for $A_i\phi$ - $player_i$ as the existential player, $opponent_i$ as the universal player. By assumption, $M \models A_i^* \phi$ (i.e. $M \models A_i \phi$) so $A_i\phi$ either succeeds or does not terminate. In particular $A_i\phi$ does not finitely fail at level n. Thus $player_i$ does not have a winning strategy. It is easy to show (see TR13 in the appendix) that, in this case, $opponent_i$ does have a winning strategy. We get $opponent_i$ to play the game according to such a strategy and we will specify how $player_i$ is to play shortly.

We note that if A_i is negative, say not(R($\bar{x}$)), then $M \models R^*(\bar{x}\phi)$ so R($\bar{x}\phi$) either finitely fails or does not terminate. In particular R($\bar{x}\phi$) does not succeed at level n. To simplify terminology we will start a game, called the ith side game, with initial goal $\leftarrow A_i\phi$. We will never play this game but now we have an ith game for each i=1...q whether A_i is positive or negative.

So let us start our game against Eloise. If A is $P(\tau)$ then $\leftarrow P(\tau\phi)$ is the first goal. Eloise will pick the literal $P(\tau\phi)$ because she has not got much choice and we find that it does unify with the head of a clause. We choose the clause $A \leftarrow A_1, \cdots, A_q$. Suppose that 0, ψ are the mgus of $P(\tau\phi)$ from $\leftarrow P(\tau\phi)$ and $P(\tau)$ from the clause (where 0 is the empty substitution since there are no variables in the first goal). Thus $\leftarrow A_1\psi, \cdots, A_q\psi$ is the next goal in the main game.

While this was going on we assume that the other games are waiting for our attention before starting. Thus the current goal in the game between $player_i$ and $opponent_i$ is $\leftarrow A_i\phi$.

From now on we map each appearance of a literal in the current goal of the main game to an appearance of a literal in the current goal of a side game. At the start we map each $A_j\psi$ to the only literal, $A_j\phi$, in the current goal of the jth game.

As we proceed we will keep the following condition satisfied.

There are substitutions $\epsilon_1, \cdots, \epsilon_q$ of some of the variables in the main current goal such that

C1) for each literal L appearing in the main current goal, if L maps into the pth game then it maps to the literal $L\epsilon_p$,

C2) if a variable symbol x appears in both literals L and K in the main current goal and L and K map into different side games, say the p1th game and the p2th game respectively, then $x\epsilon_{p1}$ and $x\epsilon_{p2}$ are the same constant, and

C3) the map is one-to-one.

At the start, since $P(\tau\phi)=P(\tau)\phi$ and 0 , ψ are mgus of $P(\tau\phi)$ and $P(\tau)$ there must be a substitution ϵ such that $\phi=\psi\epsilon$. We put $\epsilon_1, \cdots, \epsilon_q$ all equal to ϵ. Its easy to see that the conditions hold. C2 holds since each $A_j\psi\epsilon_j=A_j\phi$ is ground.

Now suppose that it is later in the game, that we have kept the condition holding and that $\leftarrow L_1, \cdots, L_p$ is the current goal in the main game. Remember that we are trying to show that we can win whatever Eloise does.

The current goal could be empty and then, in accordance with G1, we would have won. So suppose not. By G2, Eloise chooses a literal L_i and she does so according to her strategy.

Now either L_i is positive or negative.

CASE 1: L_i is positive.

In this case, we turn our attention away from the main game for a moment as proceed as follows. Since a negative literal can not be got as a substitution of a positive one our condition on the labelling implies L_i maps to a literal $L_i\epsilon_p$ in the pth game, say, and A_p is positive. So the current goal in that game is not empty and $player_p$, who has been waiting patiently for our instructions before choosing a literal, is told to choose $L_i\epsilon_p$.

Since L_i is positive so is $L_i\epsilon_p$. Furthermore, since $opponent_p$ has been playing according to a winning strategy we know that he can not lose by $L_i\epsilon_p$ failing to unify. Thus $opponent_p$ picks a clause $B \leftarrow B_1, \cdots, B_r$ which has a head which does unify with $L_i\epsilon_p$.

Suppose that the mgus of $L_i\epsilon_p$ from the last goal in the p-game and B from $B \leftarrow B_1, \cdots, B_r$ are γ , δ.

We return to main game and note that if $L_i\epsilon_p$ unifies with B then so does L_i.

Since the chosen literal does unify, by G3 Eloise does not win now.

Instead, by G4, we are to pick a clause. We choose $B \leftarrow B_1, \cdots, B_r$. Say that the mgus of L_i from $\leftarrow L_1, \cdots, L_p$ and B from the clause are θ , χ. The new current goal is

then $\leftarrow L_1\theta, \cdots, L_{i-1}\theta, B_1\chi, \cdots, B_r\chi, L_{i+1}\theta, \cdots, L_p\theta$. We define a new map as follows. If L_j (for $j \neq i$) mapped to $L_j\epsilon_p$ in the pth game then map $L_j\theta$ to $L_j\epsilon_p\gamma$ in the new pth side goal ($L_j\epsilon_p$ is not $L_i\epsilon_p$ by C3). Otherwise map $L_j\theta$ to $L_j\epsilon_{p1}$ where L_j was mapped to. Map $B_j\chi$ to $B_j\delta$ in the new pth goal.

It is easy to see that C3 is preserved.

Since θ,χ are mgus of L_i and B but $L_i\epsilon_p\gamma = B\delta$ there is a substitution ϵ'_p such that $\theta\epsilon_p' = \epsilon_p\gamma$ and $\chi\epsilon_p' = \delta$.

To define each ϵ_{p1}' for $p1 \neq p$ we proceed as follows. For each variable $x\theta$ in a literal $L_j\theta$ mapping into the p1th game, we put $x\theta\epsilon_{p1}' = x\epsilon_{p1}$.

We must check that ϵ_{p1}' is well defined. Suppose that $x\theta = y\theta$ for different symbols x , y in literals mapping into the p1th game. For the mgu θ to make $x\theta = y\theta$ means that x and y must also appear in L_i. L_i maps to the pth game, so by C2, $x\epsilon_{p1} = x\epsilon_p$ and $y\epsilon_{p1} = y\epsilon_p$ and they are both constants. So $x\epsilon_{p1} = x\epsilon_p = x\epsilon_p\gamma$ (as constants don't change under substitutions) $= x\theta\epsilon_p' = y\theta\epsilon_p' = y\epsilon_p\gamma = y\epsilon_p = y\epsilon_{p1}$. So ϵ_{p1}' is well defined.

Next we check on condition C1. Take any literal from the new current goal. There are three cases.

Case 1.1 : It is $L_j\theta$ mapping to $L_j\epsilon_p\gamma$ in the pth game. Then $L_j\theta\epsilon_p' = L_j\epsilon_p\gamma$ as required.

Case 1.2 : It is $L_j\theta$ mapping to $L_j\epsilon_{p1}$ in the p1th game. Then $L_j\theta\epsilon_{p1}' = L_j\epsilon_{p1}$ by the construction of ϵ_{p1}'.

Case 1.3 : It is $B_j\chi$ mapping to $B_j\delta$ in the pth game. Then $B_j\chi\epsilon_p' = B_j\delta$ as required.

So C1 holds.

Now we check C2. Suppose that x appears in two literals mapping to different games. There are three cases.

Case 1.1 : The literal map to games p1 and p2 neither of which are p. Since the $B_j\chi$s map to the pth game, the literals must be $L_g\theta$ and $L_h\theta$. Say $x=y\theta$ for y from L_g and $x=z\theta$ for z from L_h. L_g must have mapped to the p1th game and L_h to the p2th game. If y=z the fact that C2 held before implies that $x\epsilon_{p1}'=y\theta\epsilon_{p1}'=y\epsilon_{p1}$ is the same constant as $y\epsilon_{p2}=z\epsilon_{p2}=z\theta\epsilon_{p2}'=x\epsilon_{p2}'$. So suppose that $y\neq z$. Then, for the mgu θ to make $y\theta=z\theta$ means that both y and z appear in L_l. L_l maps into the pth game so, again since C2 held before, we have $y\epsilon_{p1}=y\epsilon_p$ is a constant. Similarly $z\epsilon_{p2}=z\epsilon_p$ is a constant. But then $y\epsilon_p=y\epsilon_p\gamma=y\theta\epsilon_p'=z\theta\epsilon_p'=z\epsilon_p\gamma=z\epsilon_p$. Thus, $x\epsilon_{p1}'=y\theta\epsilon_{p1}'=y\epsilon_{p1}=y\epsilon_p$ is the same constant as $x\epsilon_{p2}'=z\theta\epsilon_{p2}'=z\epsilon_{p2}=z\epsilon_p$.

Case 1.2 : One of the literals is $L_g\theta$ mapping to the pth game. The other literal which does not map to the pth game must be $L_j\theta$ say. Suppose $x=y\theta$ for y from L_j, $x=z\theta$ for z from L_g a, $p1\neq p$ and $L_j\theta$ maps to the p1th game. If y=z then, since C2 held before, $z\epsilon_p=y\epsilon_p$ is the same constant as $y\epsilon_{p1}$. Then $x\epsilon_p'=z\theta\epsilon_p'=z\epsilon_p\gamma=z\epsilon_p=y\epsilon_{p1}=y\theta\epsilon_{p1}'=x\epsilon_{p1}'$ which is a constant. So suppose that $y\neq z$. For θ to make $y\theta=z\theta$, y and z must appear in L_l. As before $x\epsilon_{p1}'=y\epsilon_p=y\theta\epsilon_p'=z\theta\epsilon_p'=z\epsilon_p=z\epsilon_p\gamma=z\theta\epsilon_p'=x\epsilon_p'$ is a constant.

Case 1.3 : One of the literals is $B_g\chi$. Say $x=z\chi$ for z from B_g, $x=y\theta$ for y from L_j, $p\neq p1$ and L_j maps to the p1th game. For θ,χ to make $y\theta=z\chi$, y must appear in L_1 so $x\epsilon_{p1}'=y\theta\epsilon_{p1}'=y\epsilon_{p1}=y_{\epsilon_p}$ is a constant and equals $y\epsilon_p\gamma=y\theta\epsilon_p'=z\chi\epsilon_p'=x\epsilon_p'$.

So the conditions hold.

CASE 2 : L_j is negative.

Assume that L_l is not($S(w)$). Thus the main game could be continued according to rules G5, G6 or G7. If rule G7 was applicable then we would win the main game. So let us

suppose that rule G7 is not applicable in which case L_i is ground and so equal to $L_i\epsilon_p$. Suppose further that L_i is mapped into the pth game. There are two cases.

Case 2.1) A_p is not(R(π)), so $L_i\epsilon_p$ is not(R($\pi\phi$)). We know that R($\pi\phi$) = S(w) does not succeed at level n.

Case 2.2) A_p is positive and L_i maps to $L_i\epsilon_p$ = not(S(w)) in the pth game. So the current goal in this game is not empty and *player*$_p$ is told to choose this literal. Since *player*$_p$ does not win S(w) does not succeed at level n-1 .

In both cases rule G5 is ruled out so that only rule G6 remains available for the continuance of the main game. We note that Eloise has not won during this move of the game.

The new goal is $\leftarrow L_1, \cdots, L_{i-1}, L_{i+1}, \cdots, L_p$. We keep all the mapping on the literals as before. It is easy to see that our condition on labelling is still satisfied.

Finally we need only point out that since Eloise can never win the game, either we win it at a finite stage or it goes on forever and, by G8, we win it then too.

Q.E.D. (lemma 5.4)

LEMMA 5.5

If R appears in the heads of precisely the clauses $d_1, \cdots, d_g$ then
$M \models \forall x[\bigwedge_{i=1}^{g} R^*_{d_i}(x) \rightarrow R^*(x)]$.

proof:

Suppose that ϕ is a ground substitution of x such that for each i $M \models R^*_{d_i}(x\phi)$. We must prove that R($x\phi$) either finitely fails or does not terminate. So suppose that it

succeeds, say at level n.

There is a successful n-path for $\leftarrow R(x\phi)$. The second goal will be got via a clause d such that the head of d unifies with $R(x\phi)$. Thus d is one of the d_i and $M \models R_d^*(x\phi)$.

Let us suppose that $\leftarrow A_1, \cdots, A_q$ is the body of the clause, that 0 , θ_0 are the mgus of $R(x\phi)$ from $\leftarrow R(x\phi)$ and the head from d so that $\leftarrow A_1\theta_0, \cdots, A_q\theta_0$ is the second goal in the successful path.

We choose any ground substitution ϵ for $A_1\theta_0 \cdots \theta_k, \cdots, A_k\theta_0 \cdots \theta_k$ where $\theta_1, \cdots, \theta_k$ are the mgus used in the successful path from the second step to the third and so on to the empty goal.

By the definition of the interpretation of R_d^* in M we know that there is an i=1...q such that $M \models A_i\theta_0 \cdots \theta_k\epsilon$. There are two cases.

case 1) A_i is positive, it is $P(x)$ say. So $P(x\theta_0 \cdots \theta_k\epsilon)$ either finitely fails or does not terminate. However, it is easy (but a bit messy) to extract a successful n-path for this goal from the path for $R(x\phi)$ (in which $P(x\theta)$ appears as a literal in the second step).

case 2) A_i is negative, it is not($P(x)$) say. So $P(x\theta_0 \cdots \theta_k\epsilon)$ either succeeds or does not terminate. Following the literal A_i down through the successful path for $R(x\phi)$ we must eventually find a step when the literal $A_i\theta_0 \cdots \theta_i$ is chosen. Since the path is successful $P(x\theta_0 \cdots \theta_i)$ must be ground and fail finitely at level n. The fact that $P(x\theta_0 \cdots \theta_i)\ \theta_{i+1} \cdots \theta_k\epsilon = P(x\theta_0 \cdots \theta_i)$ then gives a contradiction.

In both cases we have reached our desired contradiction.

Q.E.D. (lemma 5.5)

Q.E.D. (theorem 5.2)

5.4 Completeness of QEP with respect to SYNTH

To help us show that we can implement some logically formulated problems using NAF we can only make use of the synthesis construction if we can guarantee that in some useful cases the QEP gets all the answers that SYNTH says it should. To this end we present the completeness result of this section.

We can not hope to have a universal completeness result. In this section we show how conditions can be specified that ensure that the QEP will not flounder and in fact will go on to get all the results it should. The condition on programs - that they are covered - is a stricter form of the conditions presented in [She85] to avoid floundering. Being covered is not so strong as to be impractical, though, as we will see in the next chapter.

This may be the first generally useful completeness result for NAF to be proved. As with other results in this thesis its generalisation to the function case would be very desirable. Unlike the other results, there seems to be a bit more work involved in doing the generalisation in this case.

If **L** is a signature, **C** a finite set of constant symbols and PROG a logic program with symbols only from **L** and **C** then we say that PROG is *covered* if and only if there is a relation symbol R from **L** such that the following three conditions are satisfied.

COV1) R is in the head of clause d implies that there is a tuple $\bar{a}$ from **C** such that d is "$R(\bar{a}) \leftarrow$";

COV2) for any tuple $\bar{a}$ from **C** there is a clause "$R(\bar{a}) \leftarrow$" in PROG; and

COV3) for all clauses c from PROG, if a variable symbol x appears in a negative literal in the body of c then it also appears in a literal of the form $R(\bar{x})$ in the body of c.

We will call such an R a *grounding relation* and any positive literal containing R a *grounding literal*.

THEOREM 5.6

For all signatures **L**, for all non-empty finite sets **C** of constant symbols, for all logic programs PROG with symbols from **LUC** only, for all non-negative integers r, for all r-ary R from **L**, for all r-tuples $\bar{a}$ from **C**

if PROG is covered and SYNTH(PROG) $\vdash R(\bar{a})$ then the QEP succeeds $R(\bar{a})$.

proof:

We construct an $\mathbf{LUL}^{PROG}$ UC-structure M as follows.

1) the domain of M is the set **C** of constants;

2) each constant is interpreted as itself;

3) for each S from **L**, for each tuple $\bar{b}$ from **C**, $M \models S(\bar{b})$ if and only if the QEP succeeds $S(\bar{b})$;

4) for each S from **L**, for each tuple $\bar{b}$ from **C**, $M \models S^*(\bar{b})$ if and only if the QEP finitely fails $S(\bar{b})$;

5) for each S from **L**, for each clause d from PROG with S in the head, if the head of d is $S(\bar{v})$, the body is $A_1, \cdots, A_q$, $\bar{w}$ are the variables in $\bar{v}$ in order as in SYNTH(PROG), $\bar{z}$ the others in d in order, $\bar{x}$ all the variables in d in order then

5a) $M \models S_d^*(\bar{b})$ if and only if either

5a1) $v_i = v_j$ is a variable symbol but $b_i \neq b_j$;

5a2) v_i is the constant a but $b_i \neq a$; or

5a3) there is a ground substitution ϕ for $\bar{w}$ such that $\bar{v}\phi = \bar{b}$ and for all ground substitutions ψ for $\bar{z}$, there is i from 1 ... q such that $M \models A_i(\bar{w}\backslash\bar{w}\phi, \bar{z}\backslash\bar{z}\psi)$;

and

5b) $M \models fall_d(\bar{c})$ if and only if there is i = 1 ... q such that $M \models A_i(\bar{x}\backslash\bar{c})$.

Now M is a model of SYNTH (PROG) as

MOD1) for every clause c, $M \models POS(c)$ by lemma 5.7;

MOD2) for every R from L, M $\models$ NEG(R) because either

MOD2a) R appears in no heads when for every $\bar{b}$, R($\bar{b}$) finitely fails so M $\models \forall \bar{x} R^*(\bar{x})$; or

MOD2b) R appears in some heads when lemma 5.12 implies the result;

MOD3) for all clauses c, the definitions of interpretations of $fail_c$ and S_c^* ensure that M $\models$ NEG(c); and

MOD4) by the definition of the interpretation of the constants of C, M $\models$ INEQ(PROG).

Since SYNTH(PROG) $\vdash$ R($\bar{a}$), M $\models$ R($\bar{a}$). Thus the QEP succeeds R($\bar{a}$) as required.

It just remains to prove lemmas 5.7 and 5.12.

LEMMA 5.7

If R($\bar{x}$) $\leftarrow$ $A_1, \cdots, A_q$ is a clause in PROG then M $\models \forall \bar{x}$ ($\wedge_{i=1}^{q} A_i^* \rightarrow$ R($\bar{x}$)).

proof:

Suppose that ϕ is a ground substitution of $\bar{x}$ and for each i= 1...q, M $\models A_i^* \phi$.

For each i, we define an integer m_i as any integer such that $A_i \phi$ succeeds at level m_i if A_i is positive and such that $A_i \phi$ finitely fails at level m_i if A_i is negative . Let m be the maximum of the m_i.

We know that R($\bar{x}\phi$) and R($\bar{x}$) unify. Suppose that 0 , ψ are the mgus of R($\bar{x}\phi$) from $\leftarrow R(\bar{x}\phi)$ and R($\bar{x}$) from the clause. We show that R($\bar{x}\phi$) succeeds by constructing an m-path beginning with the goals, $\leftarrow$ R($\bar{x}\phi$) then $\leftarrow A_1\psi, \cdots, A_q\psi$.

By lemma 5.8, we can assume that all the non-ground negative $A_i\psi$s come after the others. We will continue the m-path by getting rid of each $A_i\psi$ in turn getting to goals of the form $\leftarrow A_i\psi_i, \cdots, A_q\psi_i$ every so often and eventually ending up with the empty goal. We will ensure that as i goes from 1 to q+1

C1) there is a substitution ϵ_i such that $\psi_i\epsilon_i=\phi$ and

C2) for all variables x in any $A_j\psi_i$, there is k<j such that i≤k and $A_k\psi_k$ contains x and is a grounding literal.

Let us see that these conditions hold when i=1 and ψ_1 is ψ. As $P(\sigma\phi) = P(\sigma)\ \phi$ while 0 , ψ are mgus of $R(\alpha)$ and $R(\alpha)$ we know that there is a substitution ϵ_1 such that $0\epsilon_1=0$ (which is nothing special) and $\psi\epsilon_1=\phi$. Thus C1 holds. C2 holds because of our assumed ordering of the literals and the fact that PROG is covered.

Now suppose that we have $\leftarrow A_i\psi_i, \cdots, A_q\psi_i$ as above. There are two cases.

case 1 : $A_i\psi_i$ is negative.

By C2, this first literal in the goal must be ground. So $A_i\psi_i=A_i\psi_i\epsilon_i=A_i\phi$. We know that the positive part of $A_i\phi$ finitely fails at level m so we can have a next goal $\leftarrow A_{i+1}\psi_i, \cdots, A_q\psi_i$. If we put $\psi_{i+1}=\psi_i$ and $\epsilon_{i+1}=\epsilon_i$ then we have got a goal satisfying C1. Because we have not introduced any new variable symbols in non-ground negative literals, nor got rid of any grounding literals C2 still holds.

case 2 : $A_i\psi_i$ is positive.

Since $A_i\psi_i\epsilon_i = A_i\phi$ succeeds at level m, lemma 5.9 implies that there is a substitution χ such that $A_i\psi_i$ succeeds at level m with substitution χ and $A_i\phi$ is a substitution of $A_i\psi_i\chi$.

By lemma 5.11, a path continues from $\leftarrow A_i\psi_i, A_{i+1}\psi_i, \cdots, A_q\psi_i$ to $\leftarrow A_{i+1}\psi_{i+1}, \cdots, A_q\psi_{i+1}$ in such a way that there is a substitution ϵ_{i+1} such that $\psi_{i+1}\epsilon_{i+1}=\phi$. So C1 still holds.

We have not introduced any new variables in negative literals but we may have got rid of a grounder in the form of $A_i\psi_i$. By $\bigwedge^{(cov\,1)}$ the only path got by selecting a grounder,

immediately grounds any of its variables. So any variables in non-ground negative literals which relied on $A_i\psi_i$ to conform with C2 no longer appear. So C2 still holds.

Q.E.D. (lemma 5.7)

LEMMA 5.8

If $\leftarrow L_1, \cdots, L_p$ succeeds under θ at level n and α is a permutation of { 1 , ... , p } then $\leftarrow L_{\alpha(1)}, \cdots, L_{\alpha(p)}$ succeeds under an equivalent substitution to θ at level n.

proof:

Suppose that $(G_0, \cdots, G_k)$ with $(\theta_0, \cdots, \theta_{k-1})$ is a successful n-path for $\leftarrow L_1, \cdots, L_p$ so $G_0 = \leftarrow L_1, \cdots, L_p$, $G_k = \leftarrow$ and $\theta = \theta_0 \cdots \theta_{k-1}$.

We show by induction on i= 0 ... k that there are p possibly empty lists $C_1^i, \cdots, C_p^i$ of literals, a substitution η_i of G_i and an n-path $(H_0, \cdots, H_i)$ with $(\phi_0, \cdots, \phi_{i-1})$ such that

I1) G_i is $\leftarrow C_0^i, \cdots, C_j^i$,

I2) η_i is invertible and

I3) H_i is $\leftarrow C_{\alpha(1)}^i\eta_i, \cdots, C_{\alpha(p)}^i\eta_i$.

To start with we put $H_0 = \leftarrow L_{\alpha(1)}, \cdots, L_{\alpha(p)}$, $C_i^0 = L_i$ and η_0 equal to the identity so that the conditions hold.

Now suppose that they hold for i. So $G_i = \leftarrow C_1^i, \cdots, C_p^i$. Say that K from C_j^i is selected to get to G_{i+1}. If $\alpha(r) = j$ then we select $K\eta_i$ from $C_{\alpha(r)}^i\eta_i$ in H_i to get to H_{i+1}.

There are two cases.

Case 1 : K is positive.

Say that C_j^l is $K_1, \cdots, K_s$, K is K_t and to get G_{l+1} we unify K with the head of $B \leftarrow B_1, \cdots, B_q$. Say that the mgus of K from G_l and B from the clause are θ_l, ψ_l. So G_{l+1} is

$\leftarrow C_1^l\theta_l, \cdots, C_{j-1}^l\theta_l, K_1\theta_l, \cdots, K_{t-1}\theta_l, B_1\psi_l, \cdots, B_q\psi_l, K_{t+1}\theta_l, \cdots, K_s\theta_l, C_{j+1}^l\theta_l, \cdots, C_p^l\theta_l$.

We define $C_u^{l+1} = C_u^l\theta_l$ for each u $\neq$ j and we put C_j^{l+1} equal to $K_1\theta_l, \cdots, K_{t-1}\theta_l, B_1\psi_l, \cdots, B_q\psi_l, K_{t+1}\theta_l, \cdots, K_s\theta_l$. So $G_{l+1} = \leftarrow C_1^{l+1}, \cdots, C_p^{l+1}$ as required.

Now, if K unifies with B then, as η_l is invertible, so does $K\eta_l$. Say that the mgus of $K\eta_l$ from H_l and B from the clause are ϕ_l, χ_l. Thus H_{l+1} is

$\leftarrow C_{\alpha(1)}^l\eta_l\phi_l, \cdots, C_{\alpha(r-1)}^l\eta_l\phi_l, K_1\eta_l\phi_l, \cdots, K_{t-1}\eta_l\phi_l,$

$B_1\chi_l, \cdots, B_q\chi_l, K_{t+1}\eta_l\phi_l, \cdots, K_s\eta_l\phi_l, C_{\alpha(r+1)}^l\eta_l\phi_l, \cdots, C_{\alpha(p)}^l\eta_l\phi_l$.

Now, $K\eta_l\phi_l = B\chi_l$ but θ_l, ψ_l are mgus of K and B so there is η_{l+1} such that $\eta_l\phi_l = \theta_l\eta_{l+1}$ and $\chi_l = \psi_l\eta_{l+1}$. It is straight forward to check that H_{l+1} is $\leftarrow C_{\alpha(1)}^{l+1}\eta_{l+1}, \cdots, C_{\alpha(p)}^{l+1}\eta_{l+1}$ as required.

Since $K\eta_l\eta_l^{-1}\theta_l = B\psi_l$ but ϕ_l, χ_l are mgus of $K\eta_l$ and B there is ζ_{l+1} such that $\eta_l^{-1}\theta_l = \phi_l\zeta_{l+1}$ and $\psi_l = \chi_l\zeta_{l+1}$. It is straight forward to check that $G_{l+1}\eta_{l+1}\zeta_{l+1} = G_{l+1}$ so that η_{l+1} is invertible as required.

Case 2 : K is negative.

In this case we select $K\eta_l$ from $C_{\alpha(r)}^l\eta_l$ in H_l as before. This will be ground as K is. This case is then almost a simple version of case 1.

Q.E.D. (lemma 5.8)

LEMMA 5.9

If L is a positive literal, PROG is covered, $L\phi$ succeeds under θ at level m then there is a substitution χ such that L succeeds under χ at level m and $L\phi\theta$ is a substitution of $L\chi$.

proof:

By lemma 4.4 we can construct a finished m-path tree G for $L\phi\theta$.

By lemma 4.5 we can construct an m-path $(H_0, \cdots, H_l)$ with $(\psi_0, \cdots, \psi_{k-1})$ for L which is an example of G and either H_l is all full of non-ground negative literals, its empty or we can keep going.

By lemma 5.10 H_l can not be full of non-ground negative literals. By considering sub-trees as in theorem 4.3 we know we can find a maximal such path. So it is successful.

Since it is an example of G, there is a substitution ϵ such that $L\phi\theta=H_0\psi_0 \cdots \psi_{l-1}\epsilon$. Put $\chi = \psi_0 \cdots \psi_{l-1}$ and we have our result.

Q.E.D. (lemma 5.9)

LEMMA 5.10

If L is a positive literal, PROG is covered, then no path beginning with $\leftarrow L$ can end full of non-ground negative literals.

proof:

We show by induction on the length of paths beginning with $\leftarrow L$ that any variable appearing in a negative literal in the last goal also appears in a grounding literal in that

goal.

This is true of the path ($\leftarrow L$).

Now suppose that we have a longer path. By our inductive hypothesis the condition holds of the second last goal (which is the last goal of a shorter path). Suppose that K is selected from that goal to get to the last one.

There are three cases.

Case 1 : K is positive but not a grounder.

In this case, any variables in negative literals which are descendents of literals other than K also appear in the grounders which are descendents of the appropriate grounders in the second last goal. Any variables in negative descendents of K appear in grounding descendents of K by virtue of (c.o.v.3).

Case 2 : K is a grounder.

The last goal is got from the second last one by removing K and grounding all the variables which appeared in K. Any variables left in negative literals will also appear in grounding descendents of other grounding literals in the last goal.

Case 3 : K is negative.

We get rid of a negative literal so the condition is preserved.

Q.E.D. (lemma 5.10)

LEMMA 5.11

If L_1 succeeds under θ at level m and ϵ is a substitution of $L_1, \cdots, L_p$ such that $L_1\epsilon$ is a substitution of $L_1\theta$ then there is an m-path starting with $\leftarrow L_1, \cdots, L_p$ and ending with $\leftarrow L_2\psi, \cdots, L_p\psi$ such that $L_2\epsilon, \cdots, L_p\epsilon$ is a substitution of $L_2\psi, \cdots, L_p\psi$.

proof:

Suppose that $(G_0, \cdots, G_k)$ with $(\theta_0, \cdots, \theta_{k-1})$ is a successful m-path for L_1 so that $G_0 = L_1$ and $G_k = \leftarrow$ and that $\theta = \theta_0 \cdots \theta_{k-1}$. Let ϵ be as above.

We construct a path $(H_0, \cdots, H_k)$ with $(\phi_0, \cdots, \phi_{k-1})$ such that there are lists $C_0, \cdots, C_k$ of literals, substitutions $\eta_0, \cdots, \eta_k$, and $\zeta_0, \cdots, \zeta_k$ such that

I1) $G_i = \leftarrow C_i$,

I2) $H_i = \leftarrow C_i\eta_i, L_2\zeta_i, \cdots, L_p\zeta_i$,

I3) η_i is invertible,

I4) $L_1\zeta_i = L_1\theta_0 \cdots \theta_{i-1}\eta_i$,

I5) ζ_i is invertible on the variables not in L_1

I6) if x from $L_1, \cdots, L_p$ does not appear in L_1 then $x\zeta_i$ does not appear in $C_i\zeta_i$.

At the start we put $C_0 = L_1$, η_0 and ζ_0 equal to the identity.

Suppose the conditions hold for i. Now, $G_i = \leftarrow C_i = \leftarrow K_1, \cdots, K_r$ say. Suppose that K_t is selected. We shall assume that K_t is positive because the proof when it is negative is virtually a special case of the positive proof which follows.

Suppose that K_t is unified with the head of $B \leftarrow B_1, \cdots, B_q$ with mgus θ_i, ψ_i to continue the path. So

$G_{i+1} = \leftarrow C_{i+1} = \leftarrow K_1\theta_i, \cdots, K_{t-1}\theta_i, B_1\psi_i, \cdots, B_q\psi_i, K_{t+1}\theta_i, \cdots, K_r\theta_i$.

Now H_i is $\leftarrow C_i\eta_i, L_2\zeta_i, \cdots, L_p\zeta_i$ which equals $\leftarrow K_1\eta_i, \cdots, K_r\eta_i, L_2\zeta_i, \cdots, L_p\zeta_i$.

We select $K_t\eta_t$, note that it unifies with B since η_t is invertible, and with appropriate mgus ϕ_t,χ_t we end up with H_{t+1} being $\leftarrow K_1\eta_t\phi_t, \cdots, K_{t-1}\eta_t\phi_t, B_1\chi_t, \cdots, B_q\chi_t, K_{t+1}\eta_t\phi_t, \cdots, K_r\eta_t\phi_t, L_2\zeta_t\phi_t, \cdots, L_p\zeta_t\phi_t$.

Now $K_t\eta_t\phi_t=B\chi_t$ but θ_t,ψ_t are mgus of K_t and B so there is a substitution η_{t+1} such that $\eta_t\phi_t=\theta_t\eta_{t+1}$ and $\chi_t=\psi_t\eta_{t+1}$. We put $\zeta_{t+1}=\zeta_t\phi_t$ and then it is easy to check that I2 holds for i+1.

We know that $K_t\eta_t\eta_t^{-1}\theta_t=K_t\theta_t=B\psi_t$ but ϕ_t,χ_t are mgus of $K_t\eta_t$ and B so there exists λ such that $\phi_t\lambda=\eta_t^{-1}\theta_t$ and $\chi_t\lambda=\psi_t$. Thus it is easy to check that $\eta_{t+1}\lambda$ is the identity on C_{t+1} and so I3 holds.

We know that $L_1\zeta_t=L_1\theta_0 \cdots \theta_{t-1}\eta_t$ so now we have $L_1\zeta_{t+1}=L_1\zeta_t\phi_t=L_1\theta_0 \cdots \theta_{t-1}\eta_t\phi_t=L_1\theta_0 \cdots \theta_{t-1}\theta_t\eta_{t+1}$ as required.

Now suppose that x appears in $L_2, \cdots, L_p$ but not in L_1 and that y appears in $L_1, \cdots, L_p$ but x$\neq$y. We know that $x\zeta_t\neq y\zeta_t$ and that $x\zeta_t$ does not appear in $C_t\eta_t$. Since $x\zeta_t$ does not appear in $K_t\eta_t$ then ϕ_t, which along with χ_t are mgus of $K_t\eta_t$ and B, is invertible on $x\zeta_t$. Thus $x\zeta_{t+1}\neq y\zeta_{t+1}$. This shows that I4 holds. For the same reasons $x\zeta_{t+1}$ can not appear in $C_{t+1}\eta_{t+1}$ - otherwise ϕ_t must map $x\zeta_t$ to the same symbol as it does some variable in $C_t\eta_t$. So I5 holds.

We can assume that by induction we have shown that I1-I5 hold for i=k. We put $\psi = \zeta_k$. G_k is empty so so is C_k and hence H_k is $\leftarrow L_2\psi, \cdots, L_p\psi$.

We define a substitution β on H_k as follows. If x appears in H_k then x is $y\zeta_k$ for y from $L_1, \cdots, L_p$. We put $x\beta=y\epsilon$ and we must check whether β is well defined. Suppose that $y\zeta_k=z\zeta_k$. Since ζ_k is invertible on the rest of the variables, either y=z or both y and z appear in L_1. We know that $L_1\epsilon=L_1\theta\gamma$ for some substitution γ. Then $y\epsilon=y\theta\gamma=y\theta\eta_k\eta_k^{-1}\gamma=y\zeta_k\eta_k^{-1}\gamma=z\zeta_k\eta_k^{-1}\gamma=z\theta\eta_k\eta_k^{-1}\gamma=z\theta\gamma=z\epsilon$. So β is well defined.

Thus $L_2\epsilon, \cdots, L_p\epsilon$ equals $L_2\psi\beta, \cdots, L_p\psi\beta$ as required.

Q.E.D. (lemma 5.11)

LEMMA 5.12

If R appears in precisely the heads of clauses $d_1, \cdots, d_g$ then
$M \vdash \forall x(\wedge_{i=1}^{g} R^*_{d_i}(x) \rightarrow R^*(x))$.

proof:

Suppose that R is k-ary and that ϕ is a ground substitution of x such that for all i = 1 ... g, $M \vdash R^*_{d_i}(x\phi)$. We must show that $M \vdash R^*(x\phi)$, i.e. that $R(x\phi)$ finitely fails.

For each i = 1 ... g we define an integer m_i as follows. If $R(x\phi)$ does not unify with the head of d_i put $m_i = 1$. Otherwise, suppose that d_i is $R(v) \leftarrow A_1, \cdots, A_q$, that w are the variables in v in the order as in SYNTH(PROG) and that z are the others in d_i in order. Since $M \vdash R^*_{d_i}(x\phi)$ but $x\phi$ does unify with v, we know that there is a ground substitution χ for w such that $v\chi = x\phi$ and for all ground substitutions ψ of z, there is a j such that $M \vdash A_j(w\backslash w\chi, z\backslash z\psi)$. Either A_j is positive and $A_j(w\backslash w\chi, z\backslash z\psi)$ finitely fails at level $m_{i\psi}$ say, or A_j is not(P(u)) say and $P(u)(w\backslash w\chi, z\backslash z\psi)$ succeeds at level $m_{i\psi}$ say. We put m_i to be the maximum of the (only finitely many) $m_{i\psi}$. Put m to be the maximum of the m_i.

Since PROG is covered there is a grounding relation. Pick one such and reserve the term *grounding literal* for positive literals containing it.

Now we play as the existential player against Abelard as the universal player in the m-game for $R(x\phi)$. We will win no matter how Abelard plays. By lemma 5.3 this will show that $R(x\phi)$ finitely fails as required.

THE GAME: we show how to play to win.

The current goal starts as $\leftarrow R(x\phi)$. Its not empty so Abelard does not win now. We choose the literal $R(x\phi)$ from the goal. If it does not unify then we win straight away. If it does unify then it is with some of the d_i. Suppose that Abelard chooses the clause $d = R(\tau) \leftarrow A_1, \cdots, A_q$ from the d_i. Say that the mgus of $R(x\phi)$ from $\leftarrow R(x\phi)$ and $R(\tau)$ from d are 0 , ψ where 0 is the empty substitution ($x\phi$ is ground). Thus the new current goal is $\leftarrow A_1\chi, \cdots, A_q\chi$.

As we play we will keep track of the following sequence of sets. Suppose that we have played through current goals $G_0, G_1, \cdots, G_k$ where G_0 is $\leftarrow R(x\phi)$, G_1 is $\leftarrow A_1\chi, \cdots, A_q\chi$ and G_{i+1} is got from G_i via substitution θ_i. We define the set of *possible substitutions* of G_1 at stage k by

$PS_k =$

$\{ \theta_1 \cdots \theta_{k-1}\epsilon \mid \epsilon$ is a ground substitution of precisely the variables appearing in $G_k \}$.

Now, at stage 1, this is $PS_1 = \{ \epsilon \mid \epsilon$ is a grounding substitution of the variables in $G_1 \}$. We note that this is a finite set (as C is finite and only a finite number of variables appear in G_1).

Suppose that w are the variables in the head $R(\tau)$ of d in order and z are the rest of the variables in d in order. For every ϵ from PS_1 there is a $j = 1...q$ such that $M \models A_j\chi\epsilon$. For each such ϵ we define an integer l_ϵ as follows. If A_j is positive then $A_j\chi\epsilon$ finitely fails at level m as we know from the definition of m. We choose any closed bounded set of unsuccessful m-paths for $A_j\chi\epsilon$ and let l_ϵ be any bound on the lengths of the paths. If A_j is negative we put l_ϵ be 1. We define l to be the maximum of these l_ϵs.

Suppose that the arities of the relations in L are all less than r. We will show how to play so that if the game keeps going that long then once every lr steps at the longest the set of possible substitutions after that many steps is strictly smaller than it was before.

PS_{k+1} is always included in PS_k so this shows that the game must eventually stop, for PS_k can never be empty (even if it includes only the empty substitution it is still not empty).

In fact we will show how to play so that Abelard does not win at any finite stage so that if the game must finish then it must be because of us winning.

THE OUTER LOOP : we show how to play so that in at most lr steps of the game either we win or PS is reduced.

So let us suppose that we are at stage k of the game. The previous current goals were $G_0, G_1, \cdots, G_{k-1}$ where G_0 is $\leftarrow R(x\phi)$, G_1 is $\leftarrow A_1\chi, \cdots, A_q\chi$ and each G_{i+1} is got from G_i via θ_i. PS_k is non-empty because $\theta_1 \cdots \theta_{k-1}\epsilon$ is in PS_k for any ϵ grounding the variables in G_k. We choose any such ϵ. Since $M \models R^*_d(x\phi)$ there is an i= 1...q such that $M \models A_i\chi\theta_1 \cdots \theta_{k-1}\epsilon$.
We choose any such i.

To continue the game we employ two different strategies - one if A_i is positive, the other if it is negative.

STRATEGY ONE - A_i is positive and recall that we are showing how to play to win or to reduce PS in at most lr steps.

To help us play we get some other people to play a game at the same time. As we play we will keep note of a substitution of the variables in the current goal of the main game. This is called the current substitution and is ϵ at first.

As the main game progresses we will get *player* and *opponent* to play a side game. We know that $M \models A_i\chi\theta_1 \cdots \theta_{k-1}\epsilon$ so that $A_i\chi\theta_1 \cdots \theta_{k-1}\epsilon$ finitely fails at level m. We get the two to start off playing the m-game for $A_i\chi\theta_1 \cdots \theta_{k-1}\epsilon$ - *opponent* playing the existential player according to a winning strategy (which wins within l steps) which we know exists and *player* as the universal player waiting for our instructions.

In order to use the progress of this side game for our benefit we will define a map from the literals in the current goal of the side game to literals in the past or present goals of the main game.

To help us define the map we will have to talk of descendents of appearances of literals in the goals of the main game. If G_i is $\leftarrow L_1, \cdots, L_p$ and L_j is selected to get to G_{i+1} then G_{i+1} is $\leftarrow L_1\theta_i, \cdots, L_{j-1}\theta_i, C, L_{j+1}\theta_i, \cdots, L_p\theta_i$ where either

C is $B_1\phi, \cdots, B_q\phi$, L_i is positive and the clause $B \leftarrow B_1, \cdots, B_q$ is used (with appropriate mgus θ_i , ϕ)

or C is empty, θ_i the identity and L_j is negative.

If $j \neq k$ then the *immediate descendent* of L_k from G_i is $L_k\theta_i$ from G_{i+1}. The immediate descendents of L_j from G_i are the literals, if any, in C. Descendents of appearances of literals are *descendents* of immediate descendents in the obvious transitive way.

To start off with, we map $A_i\chi\theta_1 \cdots \theta_{k-1}\epsilon$, from the beginning goal, $\leftarrow A_i\chi\theta_1 \cdots \theta_{k-1}\epsilon$, of the side game to the first descendent of $A_i\chi$ from the second goal, $\leftarrow A_1\chi, \cdots, A_q\chi$, of the main game to be selected. If no descendent of $A_i\chi$ is selected then we can trace the descendents of $A_i\chi$ down to the current main goal without one being selected and this is where we map $A_i\chi\theta_1 \cdots \theta_{k-1}\epsilon$ to.

Finally, we will keep note of a not necessarily ground substitution of the variables in the current goal of the side game. We shall call it the side substitution and start it off as the empty map (there are no variables in $A_i\chi\theta_1 \cdots \theta_{k-1}\epsilon$).

We shall soon specify some conditions on the relationship between the main game and the side game. These may or may not hold at various stages of the play. Right now in the play and at various other times we shall call a tea break. We shall see that the specified conditions hold at this first tea break and we shall see that when the game continues after a tea break at which the conditions hold then either we can play to win,

we can play to reduce PS or we can play so that another tea break can be called at which the conditions hold.

We will always have a move in the side game between tea breaks so there can be at most l tea breaks before the number of possible substitutions are strictly reduced or we win. We will have at most r moves in the main game after each tea break and before we win, we reduce PS or we have another tea break. Thus, there will be at most lr moves before we win or reduce PS as required.

Suppose that the main game has progressed through current goals $G_0, G_1, \cdots, G_k$, where G_0 is $\leftarrow R(x\phi)$, G_1 is $\leftarrow A_1\chi, \cdots, A_q\chi$ and G_{i+1} is got from G_i via substitution θ_i. Suppose further that ψ is the current substitution - a substitution of G_k - and η is the side substitution - a substitution of the variables in the current goal of the side game. We remember that we have defined a map from the literals of the current side goal to some of the past or present literals of the main game. Tea breaks will only be called when :

TB1) For every literal L from the current goal of the side game, if L maps to K from a past or present current goal G_i of the main game then $K\theta_i \cdots \theta_{k-1}\psi = L\eta$;

TB2) For every literal L from the current goal of the side game , if L maps to K of past G_i then K was chosen by us from G_i to get to G_{i+1};

TB3) For each variable x appearing as u_e in a negative literal not($P(x)$) in the main current goal there is a grounding literal, K say, containing x also in that goal such that if not($P(x)$) is mapped to by a literal not($P(y)$) in the side current goal and v_e is a variable symbol then there is a grounding literal containing v_e in the side current goal mapping to K in the main current goal;

TB4) No mapped to literal is equal to a descendant of another.

It is easy to see that the tea break conditions hold at the beginning of the side games (i.e. at the beginning of the outer loop) when the current substitution is ϵ and the side substitution is 0. TB3 holds because PROG is covered and because no negative literal is

mapped to - the only mapped to literal is the descendent of $A_i\chi$ which is positive.

THE INNER LOOP: we show how to play so that in at most r moves of the main game, either

we win,

PS is reduced or

we call another tea break.

Also, we will not call another tea break until *player* has made at least one move in the side game.

Now suppose that we have just had a tea break. We start again by inviting *opponent* to make his move in the side game. Since *opponent* is playing a winning strategy in that game the current goal there is non-empty and *opponent* picks a literal L.

There are two cases.

case 1

L maps to K of past goal G_i of the main game. So we know that K was chosen by us in going from G_i to G_{i+1}.

case 1.1

If K (and so L) is positive then Abelard must have responded with a unifying clause c with head M, say. Say that the mgus of K from G_i and M from c are θ_i , ϕ_i. So $M\theta_i=K\theta_i$. But then $M\theta_i \cdots \theta_{k-1}\psi = K\theta_i \cdots \theta_{k-1}\psi = L\eta$. Thus M unifies with L too. We get *player* to pick the clause c as a response to *opponent*'s choice of L. So the side game is continued to a new current goal and *player* has had his regulation move in the side game. Suppose that the mgus of L from the side goal and M from c are ϕ , ζ.

We must update our record of maps and substitutions. We leave the current

substitution as before - i.e as ψ.

We define a new side substitution η' as follows. Since $M\theta_l \cdots \theta_{k-1}\psi = L\eta$ but ϕ,ζ are mgus of L and M, there is a substitution η' of the new side goal such that $\phi\eta' = \eta$ and $\zeta\eta' = \theta_l \cdots \theta_{k-1}\psi$.

Now we define the new map. If $\leftarrow L_1, \cdots, L_p$ was the old side goal and $L = L_t$, c is $M \leftarrow B_1, \cdots, B_r$ then the new side current goal is $\leftarrow L_1\phi, \cdots, L_{t-1}\phi, B_1\zeta, \cdots, B_r\zeta, L_{t+1}\phi, \cdots, L_p\phi$. We map each $L_j\phi$ to where L_j was mapped to before. Now, since Abelard responded with c to our choice of K in G_l, G_{l+1} will contain $B_1\phi_l, \cdots, B_r\phi_l$. To define the image of $B_j\zeta$ under the new map we locate $B_j\phi_l$ in G_{l+1} and follow the progress of its descendants down through the history of the main game. We follow descendents of $B_j\phi_l$ down until either one is chosen in which case we map $B_j\zeta$ to this chosen literal, or we reach the current goal, in which case we map $B_j\phi$ to the descendant in the current goal.

Now, after all that work we may as well have a tea break. First we must check that we are allowed to. Clearly TB2 holds by our construction of the map. Now take any literal from the new side current goal.

The literal could be $L'\phi$ for one of the L' from the old goal. Then both $L'\phi$ and L' map to K' of G_j say. So we know that $K'\theta_j \cdots \theta_{k-1}\psi = L'\eta$. By our construction of η', though, $\phi\eta' = \eta$ so we have $K'\theta_j \cdots \theta_{k-1}\psi = L'\phi\eta'$ as required.

The other possibility is that the literal is $B\zeta$ for B from the body of c. Then $B\zeta$ maps to a later descendant, say $B\phi_l\theta_{l+1} \cdots \theta_e$ in G_{e+1}, of the $B\phi_l$ in G_{l+1}. But then, $(B\phi_l\theta_{l+1} \cdots \theta_e)\theta_{e+1} \cdots \theta_{k-1}\psi = B\zeta\eta'$ by the definition of η'.

So TB1 holds.

Let us check that TB3 holds. Suppose that x is a variable appearing as u_e in not(P(π)) in the main current goal. We have not updated the main game itself since the last tea break so there is still a grounding literal, C say, containing x in that goal. Suppose that not(P(π)) is now mapped to by a literal, not (P(τ)), in the new side goal and v_e is a variable symbol. There are two possibilities.

possibility 1 : not(P(τ)) is one of the $L_j\phi$. Then, previously, L_j = not(P(ω)) say, from the last side goal mapped to not(P(π)) and since $\tau=\omega\phi$, w_e is a variable symbol. So there was a grounding literal L_s containing w_e in $\leftarrow L_1, \cdots, L_p$ and mapping to C in the main current goal. This can not have been the L $=L_t$ selected because L mapped to a past goal. So $L_s\phi$ appears in the new side goal and maps to C. It also contains $w_e\phi=v_e$ so TB2 holds.

possibility 2 : not(P(τ) is one of the $B_j\zeta$. So not(P(π))ψ = $B_j\zeta\eta'$. By our construction of the map, in particular its action on the $B_i\zeta$s, we know that not(P(π)) is the descendant of $B_j\phi_i$ appearing in some G_{i+1} after a unification involving the clause $B\leftarrow B_1, \cdots, B_q$. In fact, not(P($\pi$)) must be $B_j\phi_i\theta_{i+1}\cdots\theta_{k-1}$. Say that B_j= not(P(ω)) so $\pi=\omega\phi_i\theta_{i+1}\cdots\theta_{k-1}$ Since u_e is a variable, w_e must also be a variable and since PROG is covered there must be B_s grounding w_e. If any of the ϕ_i or subsequent θ_{i+1} down to θ_{k-1} ground the variable appearing in the same position in each descendent of $B_s\phi_i$ then it would do the same to the other appearance of the same variable in the eth place in the descendent of $B_j\phi_i$. This can not happen, nor can the grounding of this variable by the selection of the descendent of $B_s\phi_i$ and its unification.

Thus the descendent $B_s\phi_i\theta_{i+1}\cdots\theta k$-1 appears in the main current goal with x in it. By the construction of the map $B_s\zeta$ in the current side goal maps to $B_e\phi_i\theta_{i+1}\cdots\theta_{k-1}$. Furthermore, since B_s contains the eth term w_e of B_j = not(P(ω)) then $B_s\zeta$ contains the eth term v_e of $B_j\zeta$ = not(P(τ)).

So TB3 holds.

Now we check that TB4 holds. We need only check that no $L_j\phi$ maps to a descendent of a $B_i\zeta$ or vice versa (the other checks are easy). No $L_j\phi$ maps to a descendent of a $B_i\zeta$ because then L_j previously mapped to a descendent of the selected L_t.

No $B_i\zeta$ maps to a descendent of an $L_j\phi$ because then, following $B_i\zeta$'s ancestors back, we find that the literal K of G_i which L_t mapped to is equal to, a descendent of or an ancestor of the literal which L_j (and so $L_j\phi$) maps to.

So TB4 holds.

So we can have a tea break.

case 1.2

Suppose that K is negative - say K = not(M). Since we didn't lose at G_i, M either succeeds at level m with an invertible substitution or finitely fails at level m. Since we didn't win at G_i, M must finitely fail at level m. So G_{i+1} is G_i minus K. Since *opponent* is only choosing ground negative literals L is ground. So $K\theta_i \cdots \theta_{k-1}\psi = L\eta = L$. Since M finitely fails, $M\theta_i \cdots \theta_{k-1}\psi$ does too (TR14). So the new current goal in the side game is the old one minus L. We carry the map down and keep substitutions the same. It is easy to see that we can have a tea break.

case 2 - L maps to a literal K in the main current goal

Thus the main current goal is non-empty.

case 2.1 - K is positive

We pick K in the main game. Abelard may give up and then we win. Otherwise he will pick a clause c with head M which unifies with K.

The main current goal is updated to G_{k+1} via the mgus, θ_k, ϕ_k of K from G_k and M from c.

case 2.2.1 - There is a substitution, ψ' of G_{k+1} such that $\psi=\theta_k\psi'$.

We return to the side game where *opponent* has chosen L. Now M and K unify with mgus θ_k, ϕ_k , so $M\phi_k=K\theta_k$. From TB1 we know that $K\psi=L\eta$. Thus $L\eta=K\psi=K\theta_k\psi'=M\phi_k\psi'$. Thus M and L unify.

We get *player* to pick clause c in the side game. Its continued with θ , ϕ as the mgus of L from the goal and M from c.

Now we update our records. ψ' is made the new current substitution.

We define a new side substitution η' as follows. Since θ,ϕ are mgus of L and M but $L\eta=M\phi_k\psi'$, we know that there exists a substitution η' of the new side goal such that $\theta\eta'=\eta$ and $\phi\eta'=\phi_k\psi'$.

We must also define a new map. This we do exactly as specified in case 1.1 except we carry down the map on the old main goal.

Let us check that we can have a tea break. Clearly, TB2 holds.

If we have a literal from the new current goal of the side game then it is either $L\theta$ for L from the old goal or it is $B\phi$ for B from the body of c.

If it is $L\theta$ then $L\theta$ and L map to K of G_i say. We know that $K\theta_i \cdots \theta_{k-1}\psi = L\eta$. Thus $K\theta_i \cdots \theta_{k-1}\theta_k\psi' = K\theta_i \cdots \theta_{k-1}\psi = L\eta = L\theta\eta'$.

If it is $B\phi$ then it maps to $B\phi_k$ in G_{k+1} and $B\phi_k\psi' = B\phi\eta'$. Thus TB1 holds.

Now we show that TB3 holds. Say that the old main goal was $\leftarrow K_1, \cdots, K_p$ and K=K_t was selected. The new main goal is

$\leftarrow K_1\theta_k, \cdots, K_{i-1}\theta_k, B_1\phi_k, \cdots, B_q\phi_k, K_{i+1}\theta_k, \cdots, K_p\theta_k$. Suppose that variable x appears as u_i in not(P ($\bar{x}$)) in the new main goal. There are two cases.

case 1 : not(P ($\bar{x}$)) = $K_j\theta_k$. Say K_j = not(P($\bar{w}$)). Then w_i is not ground so there is a grounding literal containing w_i as one of the K_rs. It can not be the chosen K_i because then (by the properties of grounding literals) θ_k would ground w_i (but $w_i\theta_k = u_i$ = x). So $K_r\theta_k$ appears containing x in the new goal. If $K_j\theta_k$ is mapped to by a literal with non-ground ith argument in the new side goal then this literal is one brought down from the previous goal as each $B_r\phi$ maps to $B_r\theta_k$. So its immediate ancestor mapped to K_j. Thus there was a grounder for its ith argument also in the side goal and mapping to a grounder in the main goal. If that grounder was L then K was a grounder for w_i. Thus θ_k grounds w_i contrary to our supposition. Thus this grounder was not L and its descendent is still in the side goal.

case 2: not(P($\bar{x}$)) = $B_r\phi_k$. Say B_r = not(P($\bar{w}$)), $u_i = w_i\phi_k$ is the variable x so then w_i is a variable. Since PROG is covered, there is a B_s grounding w_i. So $B_s\phi_k$ is a grounder for u_i. Also $B_r\phi$ maps to $B_r\phi_k$ and $B_s\phi$ maps to $B_s\phi_k$, $B_s\phi$ being a grounder for the ith argument in $B_r\phi$.

Now to check TB4. This is easy as no $B_r\phi_k$ is a descendent of (or equal to) some other mapped literal , because its immediate ancestor, K, was not.

case 2.2.2 - there is no substitution ψ' of G_{k+1} such that $\psi = \theta_k\psi'$.

We note that the number of possible side games is reduced as $\theta_1 \cdots \theta_{k-1}\psi$ which is in PS_k is not in PS_{k+1}.

case 2.2 - K is negative

We would like to pick K in the main game. However, K may not be ground and so we may have to do a bit of work beforehand. For each variable in K in some order we

repeat the following procedure to ground the variable.

If x appears in K then by TB3 there is a grounder for x in the main goal. We pick that. Abelard will respond with one of the grounding clauses. If the mgu θ_k does not agree with the current substitution - i.e. there is no substitution ψ' of G_{k+1} such that $\theta_1 \cdots \theta_{k-1}\psi = \theta_1 \cdots \theta_{k-1}\theta_k\psi'$ - then, as in case 2.1.2, we have reduced the number of possible substitutions of G_1 and we have completed our task.

Otherwise, we carry down the maps, make such a ψ' the new current substitution. We do not need to make a new side substitution because we have not extended the game and for each mapping L', if it maps to K' in G_i then $K'\theta_i \cdots \theta_{k-1}\psi = L'\eta$ so $K'\theta_i \cdots \theta_k\psi' = L'\eta$.

If we have not yet grounded $K\theta_k$ by now we repeat the procedure again for each of the remaining variables. We can only do this at most r times before we have grounded K.

Now that K (or its substituted descendent) is ground we choose it in the main game. Since we have kept the tea break conditions holding $K = K\psi = L\eta$. Now *opponent*, who is playing a winning strategy, chose L in the side game so it must be ground and equal to K and the positive part of L must either succeed or finitely fail . In the first case, the positive part of K succeeds so we win the main game.

In the second case, the next current goal in both games is the old one minus the chosen literal.

We keep ψ and η as before and map each literal in the new side current goal to where it was mapped before, except those mapping to the old main current goal get mapped to the corresponding literal in the new current goal.

Clearly, since θ_k is the identity, we can have a tea break.

STRATEGY TWO - A_i is negative and recall that we are showing how to play to win or to reduce PS in at most lr steps.

Suppose that A_i is not($P(\pi)$). Thus $P(\pi)\chi\theta_1 \cdots \theta_{k-1}\epsilon$ succeeds at level m.

A successor of the literal $A_i\chi$ from G_1 must still appear in G_k for otherwise we must have chosen the literal not($P(\pi\chi\theta_1 \cdots \theta_j)$) from a past current goal G_{j+1}. If this had happened, since $P(\pi\chi\theta_1 \cdots \theta_{k-1}\epsilon)$ is a substitution of $P(\pi\chi\theta_1 \cdots \theta_j)$, the latter would not have finitely failed contrary to the evidence that we are still playing the game.

So $A_i\chi\theta_1 \cdots \theta_{k-1}$ appears in the current goal G_k.

If it is not ground then we can, by lemma 5.10, choose a grounder for any variable x appearing in it. Abelard will respond with a grounding clause and we will get a new mgu θ_k. Perhaps PS_{k+1} is strictly smaller than PS_k in which case we have completed our task. Otherwise $\chi\theta_1 \cdots \theta_{k-1}\epsilon$ is in PS_{k+1} , say that $\chi\theta_1 \cdots \theta_{k-1}\epsilon = \chi\theta_1 \cdots \theta_{k-1}\theta_k\eta$. So $P(\pi)$ $\chi\theta_1 \cdots \theta_k\eta$ succeeds at level m. If $A_i\chi\theta_1 \cdots \theta_k$ is not ground we repeat the procedure. We can only do this at most r times.

So we may assume that $A_i\chi\theta_1 \cdots \theta_j$ is ground and $P(\pi)$ $\chi\theta_1 \cdots \theta_j\eta$ succeeds at level m. But since $P(\pi)$ $\chi\theta_1 \cdots \theta_j$ is ground it equals $P(\pi)$ $\chi\theta_1 \cdots \theta_j\eta$. So we choose $A_i\chi\theta_1 \cdots \theta_j$ and clearly we win the game.

Q.E.D.

Chapter 6

LPNAF and DEF

In this chapter we will demonstrate the close relationship between the queries implementable using logic programming with negation as failure (LPNAF) and those expressible in the formal language DEF. In fact, we will show that not only are they the same class of queries but that it is easy to convert from a logic program implementing a query to the DEF expression of the query and vice versa.

To do this we first use the completeness result for NAF to show how to find an LPNAF program which completely implements each DEF query. By exploiting the SYNTH construction, establishing this result is mainly a matter of technicalities.

For the dual result, showing that DEF expresses LPNAF, we use the correctness result for NAF. We show how to construct a DEF query which is correctly implemented by each LPNAF program.

6.1 Implementing DEF

Let us start to make some use of the synthesis construction now by showing how to implement any DEF query in a logic program with NAF.

We saw how to implement any Datalog query by sticking a logic program after the logic programming description prog(D), as defined below, of the queried structure D.

DEFINITION

If D is a finite L-structure then prog(D) is the following logic program consisting only of atomic facts with relation symbols from L and constant symbols the elements of D. For each non-negative integer k, for each k-ary symbol R from L, for each k-tuple $\bar{a}$ from D the clause "$R(\bar{a}) \leftarrow$" is in prog(D) if and only if $D \models R(\bar{a})$.

LPNAF has loosely been defined as the generalisation of this method of query answering to the case when impure programs can be added to a description of the queried structure. Unfortunately there is a major problem with this approach if we continue to describe each queried structure D by prog(D). When it comes to non-Horn-clause queries the description is not adequate - structures with different answers to the same query can have the same description. Elements which do not appear in any tuples in the extensions of L relations do not even appear in the program. It is not hard to see that queries involving negations or universal quantifications of relations from L can not be answered on the basis of prog.

If we want to answer more general queries we shall have to improve the description of the queried structure. In fact, it is very easy to augment the description prog to make it adequate for other queries. We just need to describe the domain of the queried structure.

DEFINITION If we are going to be querying L-structures then we choose a 1-ary relation symbol, *exist*, not in L. For each finite L-structure D, progex(D) is the logic program with relation symbols from $L \cup \{exist\}$, constant symbols the elements of D and containing prog(D) plus all the clauses of the form "exist(a) $\leftarrow$" for a from D.

Clearly, we have now described each structure up to isomorphism- i.e. more

adequately than we need for the type of queries we are asking. Now, let us see if we can use this method for implementing any DEF query. Suppose that PROV(T;Q) is such a query. If **L** and **S** are disjoint signatures, *exist* is a 1-ary relation symbol not from **L** nor **S** and **T** is a definite (L,S)-implicational theory then we construct the logic program implementing **T** as follows.

First, distinct 2-ary relation symbols *eq* and *neq* are chosen which are not in $\mathbf{L} \cup \mathbf{S} \cup \{exist\}$.

Second, for each sentence ϕ from **T**, for each conjunct c from the body of the sentence, if the conjunct is $\forall y S(\bar{x})$ then a new relation symbol $S_{\phi c}$ is chosen. If $\bar{x}$ contains the distinct variable symbols $x_1, \cdots, x_l$ apart from y then $S_{\phi c}$ is l-ary. Let **R** be the signature containing **L**, **S** , *exist* , *eq* , *neq* and all these new relation symbols.

We define a program PROG with relation symbols from **R** and no constant symbols.

A) For each sentence $\phi = \forall z[\wedge_{i=1}^{q} A_i \rightarrow A]$ from **T**,

1) put $A \leftarrow [A_1], \cdots [A_q]$ in PROG, where, for each conjunct $c = A_i$ from the body of the implication,

1.1) if R is from **L** or **S** then [$R(\bar{x})$] is $R(\bar{x})$,

1.2) [$u \neq v$] is *neq*(u,v),

1.3) if R is from **L** and $\bar{x}$ contains distinct variable symbols $x_1, \cdots, x_l$ then [$\neg R(\bar{x})$] is $exist(x_1), \cdots, exist(x_l), not(R(\bar{x}))$,

1.4) if S is from **S** and $\bar{x}$ contains distinct variable symbols $x_1, \cdots, x_l$ apart from y then [$\forall y S(\bar{x})$] is $exist(x_1), \cdots, exist(x_l), not(S_{\phi c}(x_1, \cdots, x_l))$.

B) For each sentence ϕ from **T**, for each conjunct c in the body of the implication, if c is $\forall y S(\bar{x})$, $\bar{x}$ contains distinct variable symbols $x_1, \cdots, x_l$ apart from y then put $S_{\phi c}(x_1, \cdots, x_l) \leftarrow exist(x_1), \cdots, exist(x_l), exist(y), not(S(\bar{x}))$ in PROG.

C) Put $eq(x,x)\leftarrow$ and $neq(x,y)\leftarrow exist(x),exist(y),not(\ eq(x,y)\ )$ in PROG.

The implementation of the popular brother query in section 1.7 uses exactly this construction. The fact that that example works is not surprising given the following result.

LEMMA 6.1

For any signature **L**, for any n-ary query PROV(T;Q) from DEF(**L**), if PROG is the logic program constructed from **T** in the manner described above then for all finite **L**-structures D, for all σ from D^n,
progex(D) + PROG succeeds $Q(\sigma)$ if and only if D $\models$ PROV(T;Q) (σ).

proof:
part one)

Suppose that progex(D) + PROG succeeds $Q(\sigma)$.

Put SYNTH(progex(D) + PROG) = SYNTH(progex(D) + PROG , **R** , **C**). Thus, SYNTH(progex(D) + PROG) $\vdash Q(\sigma)$. We must prove that D$\models$ PROV(T;Q) (σ). This will be accomplished by showing that any model M of FA(D)$\cup$**T** is a model of $Q(\sigma)$ and using result 2.2.

First we construct a model N of SYNTH(progex(D) + PROG) from M. We will then know that N is a model of $Q(\sigma)$. However, our construction will be such that M will be a model of $Q(\sigma)$ too.

If **T** is an **L**$\cup$**S**-theory and **C** is the finite set of constant symbols being the domain of D in a different guise, then M is an **L**$\cup$**S**$\cup$**C**-structure.

Now, progex(D) is a program with relation symbols from $\mathbf{L} \cup \{exist\}$ and constant symbols from **C**, PROG is a program with no constant symbols and with relation symbols from $\mathbf{R} = \mathbf{L} \cup \mathbf{S} \cup \mathbf{S}_T \cup \{exist, eq, neq\}$ where $\mathbf{S}_T$ contains all the extra relation symbols introduced by the construction of PROG from **T**. Thus SYNTH(progex(D) + PROG) will be a $\mathbf{R} \cup \mathbf{R}^* \cup \mathbf{R}^*_{progex(D)+PROG} \cup fall_{progex(D)+PROG} \cup \mathbf{C}$-theory. We define a structure N from these symbols.

1) The domain of N is that of M.

2) The constants of **C** are interpreted in N as in M. Since M is a model of FA(D) it is a model of DCA(D) and UNA(D) so we might as well regard its domain, and so the domain of N, as being the constants **C**, i.e. the domain of D.

3) **L** and **S** are interpreted on N as on M.

4) *exist* is the full 1-ary relation.

5) *neq* is the not equal 2-ary relation on N.

6) For R from **L**, R^* is the complement of R on N.

7) eq^* is the same as *neq*.

8) For each sentence ϕ from **T**, for each conjunct c of the form $\forall y S(\bar{x})$, if $\bar{x}$ are the l distinct variables apart from y in $\bar{a}$ (as chosen and ordered in the construction of PROG) then for every l-tuple $\bar{a}$ from N, $N \models S_{\phi c}(\bar{a})$ if and only if $M \models \neg \forall y S(\bar{a}(\bar{x},y))$.

9) For each R from **L**, for each clause c of progex(D) with R in the head, $N \models R^*_c(\bar{a})$ if and only if c is not "$R(\bar{a}) \leftarrow$".

10) If d is the clause with *eq* in its head then eq^*_d is the not equal 2-ary relation on N.

11) If d is the clause with $S_{\phi c}$ in its head then $S^{*}_{\phi c,d}$ is the same relation as $S_{\phi c}$.

12) *exist** is the empty 1-ary relation.

13) If d is the clause with $S_{\phi c}$ in its head then (assuming the SYNTH construction took the variables x , y of d in that order) we put N$\models$ $fall_d(a,b)$ if and only if M$\models S(a(a,b)$.

14) If d is a clause from progex(D) then $fall_d$ is the false 0-ary relation.

15) If d is the clause "eq(x,x) $\leftarrow$", then $fall_d$ is the empty 1-ary relation.

16) The rest of the relation symbols do not matter too much and we interpret them as full relations of the appropriate arity.

Now we must establish our claim that N is a model of SYNTH(progex(D) + PROG). This is straight forward to do sentence by sentence if a little time and space consuming. Thus, we must show that N is a model of POS(d) for each clause d from progex(D) or PROG, NEG(P) for each symbol P from L$\cup$S$\cup$S$_T\cup$ {*exist* , *eq* , *neq* }, NEG(d) for each clause d and INEQ(C).

Most of these sentences require very simple proofs. We will just look at two of the hardest.

Suppose that d is the clause $A \leftarrow [A_1], \cdots [A_q]$ from PROG and that we must show that N$\models POS(d)$. So T contains the sentence $\forall z(\wedge_{i=1}^{q} A_i \rightarrow A)$, we know M$\models$T and we want to show that N $\models \forall z(\wedge_{i=1}^{q} [A_i]^{*} \rightarrow A)$. Suppose that a is from N such that for each i= 1...q N$\models [A_i]^{*}(z\backslash a)$. We will show that for each i=1..q M$\models A_i(z\backslash a)$. Thus, M$\models A(z\backslash a)$ and since M and N have the same interpretations for symbols from S, N$\models A(z\backslash a)$ as required.

If A_i is R(a) where R is from L or S then $[A_i]$ is A_i so $[A_i]^{*}$ is A_i. Thus N$\models R(a(z\backslash a))$ and M$\models A_i(z\backslash a)$ too because N and M agree on R.

If A_i is $u \neq v$ then $[A_i]^+$ is $neq(u,v)$. So N⊨$neq(u,v)(z\backslash a)$. This means that u($z\backslash a$) is a different element of the domain of N from v($z\backslash a$). So they are different in M too and M⊨$A_i(z\backslash a)$.

If A_i is $\neg R(a)$ with R from L, a containing $x_1, \cdots, x_l$ then $[A_i]^+$ is $exist(x_1)\wedge \cdots \wedge exist(x_l)\wedge R^*(a)$. So N⊨$R^*(a(z\backslash a))$. By construction of N this means M⊨$\neg R(a(a))$ so M⊨$A_i(z\backslash a)$.

If A_i is $\forall y S(a)$, S from S, a containing $x_1, \cdots, x_l$ and y then $[A_i]^+$ is $exist(x_1)\wedge \cdots \wedge exist(x_l)\wedge S^*_{\phi c}(x)$. So N⊨$S^*_{\phi c}(x(z\backslash a))$. By construction of N this means M⊨$\forall y S(a,y)(z\backslash a)$, i.e. M⊨$A_i(z\backslash a)$.

This completes our proof that N⊨$POS(d)$.

The other proofs involved in showing that N⊨ SYNTH(progex(D) + PROG) are all much easier. The only other proof involving a bit of thought is the following.

Suppose that d is the clause "$S_{\phi c}(x) \leftarrow exist(y), not(\ S(a)\ )$" and that we must show that N⊨$NEG(d)$. NEG(d) contains three sentences:

a) $\forall x((\forall y\ fall_d(x,y) \rightarrow S^*_{\phi c,d}(x))$ which is true in N because of the interpretations of $S^*_{\phi c,d}$ and of $fall_d$;

b) $\forall xy(exist^*(y) \rightarrow fall_d(x,y))$ which is true because $exist^*$ is empty; and

c) $\forall xy(S(a) \rightarrow fall_d(x,y))$ which is true because of the interpretation of $fall_d$ being defined as it is.

This completes what is worth seeing of the proof of part one of our main theorem for this section.

part two)

Suppose that D $\models$ PROV(T;Q) (σ). Then FA(D)$\cup$T $\vdash Q(\sigma)$. We show that SYNTH(progex(D) + PROG) $\vdash Q(\sigma)$. So then, by theorem 5.6, $Q(\sigma)$ succeeds from progex(D) + PROG as progex(D) + PROG is covered.

To proceed, we suppose that N is a model of SYNTH(progex(D) + PROG). Note that N may be an infinite structure. We use N to construct a model, M, of FA(D)$\cup$T. Thus M is a model of $Q(\sigma)$ and our construction will be such that this implies that N is a model of $Q(\sigma)$ as well.

If **C** is the domain of D taken as a set of constant symbols and **S** is the signature of new relations in **T** then M is the **L**U**S**U**C**-structure with the same domain as D, interpretations of the constants from **C** as themselves, interpretations of symbols from **L** as in D and interpretations of symbols from **S** as in N i.e. M$\models S(\sigma)$ if and only if N$\models S(\sigma)$.

It is clear that M$\models$FA(D) and that if M$\models Q(\sigma)$ then N$\models Q(\sigma)$ so it only remains for us to prove that M$\models$**T**.

Given any sentence $\forall z(\wedge_{i=1}^{j} A_i \rightarrow A)$ from **T** we shall show that M is a model of it. Suppose that α is from M (i.e. from D) such that M is a model of each of the $A_i(z\backslash\alpha)$. We know that N is a model of $\forall z(\wedge_{i=1}^{j}[A_i]^+ \rightarrow A)$. If we can show that N is a model of each $[A_i]^+$ then we will know that N$\models A(z\backslash\alpha)$. But N and M agree on the relation from S which appears in A so M$\models A(z\backslash\alpha)$ as required.

If A_i is R(α) with R from **L** then M$\models A_i(z\backslash\alpha)$ implies that $D \models R(\alpha(\alpha))$. So the clause asserting this is in progex(D). SYNTH(progex(D) + PROG) contains R($\alpha(\alpha)$) so N is a model of it. Hence, N$\models [A_i]^+(z\backslash\alpha)$ as required.

If A_i is S(α) with S from **S** then M$\models S(\alpha(\alpha))$. But M and N agree on S so N$\models [A_i]^+(z\backslash\alpha)$ as required.

If A_i is $z_p \neq z_q$ then M$\models a_p \neq a_q$. So ($a_p \neq a_q$) is in SYNTH(progex(D) + PROG). Thus N$\models a_p \neq a_q$. If d is the clause "eq(x,x) ←" from PROG then the sentence

$\forall uv(u \neq v \rightarrow eq_d^*(u,v))$

is in SYNTH(progex(D) + PROG). N is a model of this so it is also a model of $eq_d^*(a_p,a_q)$. Since d is the only clause with eq in its head N is also a model of

$\forall xy(eq_d^*(x,y) \rightarrow eq^*(x,y))$

and hence of $eq^*(a_p,a_q)$. We also know that N is a model of $exist(a_p)$ and $exist(a_q)$ because of the appropriate clauses in progex(D). Finally, since " neq(x,y) ← exist(x) , exist(y) , not(eq(x,y))" is in PROG, we know that N is a model of $\forall xy(exist(x) \wedge exist(y) \wedge eq^*(x,y) \rightarrow$ neq$(x,y))$. Putting this all together we can deduce that N$\models neq(a_p,a_q)$, i.e. N $\models [A_i]^+(\bar{z} \backslash \bar{a})$, as required.

If A_i is $\neg R(\bar{a})$ for R from **L** and $\bar{a}(\bar{z} \backslash \bar{a})$ is $\bar{b}$ then M$\models \neg R(\bar{b})$ so $D \models \neg R(\bar{b})$. Thus R($\bar{b}$) does not appear in progex(D). If R appears in the heads of no clauses then N$\models R^*(\bar{b})$ (because its a model of the sentence $\forall \bar{x} R^*(\bar{x})$ from the synthesis) and otherwise N$\models R^*(\bar{b})$ (because we can go through all the clauses d from progex(D) with R in their heads showing that N is a model of $R_d^*(\bar{b})$ and then use the sentence relating all the R_d^* s to R^*). Finally, we can show that N is a model of each $exist(x_i)(\bar{z} \backslash \bar{a})$ for each x_i appearing in $\bar{a}$ so N is a model of $[A_i]^+(\bar{z} \backslash \bar{a})$, i.e. $exist(x_1) \wedge \cdots \wedge (x_l) \wedge R^*(\bar{a})(\bar{z} \backslash \bar{a})$.

If A_i is $\forall y S(\bar{a})$ with S from **S** then M$\models \forall y S(\bar{a}(\bar{a},y))$. Suppose that ϕ is the sentence from **T** that A_i is the conjunct c from the body of. Suppose further that d is the clause "$S_{\phi c}(\bar{x}) \leftarrow exist(y)$, not($S(\bar{a})$)". If $\bar{x}$ is $x_1, \cdots, x_l$ let us put $x_j(\bar{z} \backslash \bar{a}) = e_j$. We aim to show that N$\models \forall y\, fail_d(\bar{e},y)$. Given any element n from the domain of N there are two possibilities.

If n is the interpretation of b from **C** then, since M$\models S(\bar{a}(\bar{a},b))$, by the construction of M from N we know that N$\models S(\bar{a}(\bar{a},b))$. But (check NEG(d)) N is a model of the sentence $\forall \bar{x} y(S(\bar{a}) \rightarrow fail_d(\bar{x},y))$ so N$\models fail_d(\bar{e},b)$.

If n is not the interpretation of any constant from C then, for each clause cl = " exist(b) ←" from progex(D) + PROG with exist in its head, we have $N \models \forall x(x \neq b \rightarrow exist^{*}_{cl}(x))$.

So N is a model of $exist^{*}_{cl}(n)$. However $N \models \forall x(\bigwedge_{\text{all such } cl} exist^{*}_{cl}(x) \rightarrow exist^{*}(x))$. So N is a model of $exist^{*}(n)$. But (check NEG(d) again) N is a model of $\forall xy(exist^{*}(y) \rightarrow fail_{d}(x,y))$ so $N \models fail_{d}(\bar{e},n)$.

So $N \models \forall y\, fail_{d}(\bar{e},y)$ as claimed. Because the sentence

$\forall x((\ \forall y\, fail_{d}(x,y)\ ) \rightarrow S_{\phi c,d}(x))$

is in NEG(d) we know that $N \models S_{\phi c,d}(\bar{e})$. Now, by the construction of PROG, d was the only clause with $S_{\phi c,d}$ in its head so $N \models \forall x(S_{\phi c}(x) \rightarrow S_{\phi c}(x))$ and hence $N \models S_{\phi c}(\bar{e})$. Also, each of the e_js are in C so $N \models exist(e_j)$. Thus N is a model of $exist(e_1) \wedge \cdots \wedge exist(e_l) \wedge S_{\phi c}(\bar{e})$, i.e. $[A_i]^{+}(\bar{x} \backslash \bar{a})$.

This completes the proof of the second part of the theorem.

Q.E.D.

6.2 Expressing LP with NAF queries in DEF

In this section we show that each query which can be implemented by tacking a function-free and constant-free logic program with NAF onto the description progex(D) of the queried structure D is expressible as a DEF query. Most of the work has already been done when we use the synthesis to represent the program. We know that a synthesis looks like a DEF theory with constants in it.

Once we have shown how to construct a DEF theory from the program the proof that they represent the same query is much the same as the proof in the last section. We will give even less of the details.

So suppose that we have a logic program PROG which we use to query finite L-structures D in conjunction with progex(D). Our first step is to show that we might as well be using a program with relation symbols from **L** not appearing in the heads of clauses. This is because we can always introduce new derived relation symbols R' for each R from $\mathbf{L} \cup \{exist\}$, change the appearances of R in the heads of clauses to R', change the appearances of R in the body of clauses to R' and add clauses of the form
" $R'(x) \leftarrow R(x)$".
It is easy to see that this program implements the same query as does PROG.

So, without loss of generality we can assume that PROG has no appearances of symbols from $\mathbf{L} \cup \{exist\}$ in the heads of clauses.

The second step is to form the synthesis of PROG assuming that there are no constant symbols in its language (there are none in the program itself) and that the only relation symbols are those from $\mathbf{L} \cup \{exist\}$ and those extra ones appearing in PROG.

From the synthesis, we remove the sentences
1) $\forall x R'(x)$ where R is from **L** and
2) $\forall x \; exist'(x)$.
We add the extra sentences
1) $\forall x(\neg R(x) \rightarrow R'(x))$ for each R from **L**.
Call the resulting theory **P**.

P is not quite a DEF theory because it may contain multiple nestings of universal quantifications in the bodies of implications. It is a simple matter to add extra symbols and sentences to get an equivalent but DEF theory. We shall prove all our results without loss of generality by assuming that **P** is itself a DEF theory.

Let us call the signature of non-L symbols in PROG, **S** and the signature of derived relation symbols in **P**, **T**. Thus **S** is contained in **T**.

LEMMA 6.2

For all finite L-structures D, for all non-negative integers r , for all r-ary Q from S, for all r-tuples c from D,

1) if progex(D) + PROG succeeds $Q(c)$ then $D \models PROV(P;Q)\ (c)$ and

2) if progex(D) + PROG finitely fails $Q(c)$ then we do not have $D \models PROV(P;Q)\ (c)$.

proof:

part one)

First of all we show that we can expand any model of FA(D)U **P** into a model of SYNTH(progex(D) + PROG). Suppose that M is a model of FA(D)U **P**. To expand M to N we will define interpretations for the symbols R_d^* for R from **L** and d a clause from progex(D), and for symbols $fall_d$ for d from progex(D). Since M is a model of FA(D) we may as well assume that its domain is that of D.

For each clause d from progex(D), for each k-ary R from **L** , for each k-tuple $\bar{b}$ from D we put

1) $N \models R_d^*(\bar{b})$ if and only if d is not " $R(\bar{b})\leftarrow$" and

2) we make $fall_d$ the 0-ary false relation.

We can show sentence by sentence that N is a model of

SYNTH(progex(D) + PROG).

This can be done in such a straight forward manner that it is not worth presenting here.

part two)

Suppose that progex(D) + PROG succeeds $Q(c)$. Thus,

SYNTH(progex(D) + PROG) $\vdash Q(c)$.

If M is any expansion of D to a model of **P** then we know, by part one, that M can be

expanded to a model N of SYNTH(progex(D) + PROG). Then we know that N and hence M is a model of Q(σ). This shows that D $\models$ PROV(P;Q) (σ) as claimed in the first part of the theorem.

part three)

Suppose that progex(D) + PROG finitely fails Q(σ). Then by part b of the proof of theorem 5.1 we know that SYNTH(progex(D) + PROG) $\vdash Q'(\sigma)$. If M is any expansion of D to a model of P, we know by part one, that M can be expanded to a model N of SYNTH(progex(D) + PROG). Thus N$\models$ $Q^*(\sigma)$. Since Q^* is interpreted on the reduct M of N as in N, M$\models$ $Q^*(\sigma)$. Thus D $\models$ PROV(P;Q*) (σ).

Our task will have been completed if we now show that we can not have both D $\models$ PROV(P;Q*) (σ) and D $\models$ PROV(P;Q) (σ).

To do this it seems easiest to hark back to our results relating DEF to $\overline{LFP}$. We have shown how to construct an L-prescription r such that for each S from S, PROV(P; S) = $\overline{LFP}$ (r ; S). Let us suppose that r prescribes F on D.

Assume for contradiction that D $\models$ PROV(P;Q) (σ) and D $\models$ PROV(P;Q*) (σ). Thus $\sigma \in [F^m(\varnothing)]_Q$ and $\sigma \in [F^m(\varnothing)]_{Q^*}$ for large enough m. Hence we can choose the least k such that there is S from S with $[F^k(\varnothing)]_S \cap [F^k(\varnothing)]_{S^*}$ nonempty.

Let us suppose that $\bar{b} \in [F^k(\varnothing)]_S$ and $\bar{b} \in [F^k(\varnothing)]_{S^*}$.

Looking at the construction of r, we see that since D $\models \phi_S(F^{k-1}(\varnothing),\bar{b})$ there is an implication $\forall x(\wedge_{i=1}^{q} A_i \rightarrow S(v))$ in P and a tuple $\bar{a}$ such that $\bar{b}=v(\bar{a})$ and
D $\models \wedge_{i=1}^{q} A_i(F^{k-1}(\varnothing),\bar{a})$.
We note that then $S(v) \leftarrow A_1, \cdots, A_q$ is a clause in PROG.

Now since D $\models \phi_{S^*}(F^{k-1}(\varnothing),\bar{b})$ and $\forall x(\wedge_{i=1}^{s} S^*_{d_i}(x) \rightarrow S^*(x))$ is the only sentence in P

with head containing S^*, we must have $D \models [\bigwedge_{i=1}^{q} S^*_{d_i}(\bar{b})](F^{k-1}(\varnothing))$.

Since d = " $S(\bar{v}) \leftarrow A_1, \cdots, A_q$ " is one of the clauses $d_1, \cdots, d_g$ with heads containing S, we have $D \models S^*_d(\bar{b})(F^{k-1}(\varnothing))$.

Since $S^*_{d_i}$ is not in L this means that k-1 is greater than zero and that $D \models \phi_{S^*_d}(F_{k-2}(\varnothing), \bar{b})$. Looking at the construction of $\phi_{S^*_d}$ from P we see that it is a disjunction of formulae - one corresponding to each sentence in P with S^*_d in the head. Since $\bar{b} = \bar{v}(\bar{a})$, D is not a model of the disjuncts corresponding to the sentences $\forall x(x_i \neq x_j \rightarrow S^*_d(x))$ nor the sentences $\forall x(x_i \neq a \rightarrow S^*_d(x))$. Thus $D \models \forall z\, fall_d(x)(F^{k-2}(\varnothing), \bar{e})$ where $\bar{w}$ are the free variables in $\bar{v}$ and $\bar{e} = \bar{w}(\bar{a})$.

Choosing a substitution for $\bar{z}$ which, together with $\bar{c}$ for $\bar{w}$ maps $\bar{x}$ to $\bar{a}$ we find that $D \models fall_d(\bar{a})(F^{k-2}(\varnothing))$. Thus k-2 is greater than zero and $D \models \phi_{fall_d}(F^{k-3}(\varnothing), \bar{a})$. Looking at the construction of this from P we find that it is a disjunction of A_i. So there is an i = 1 ... q such that $D \models A_i(F^{k-3}(\varnothing), \bar{a})$ and since F is monotone $D \models A_i(F^{k-1}(\varnothing), \bar{a})$.

We have already shown that $D \models A_i(F^{k-1}(\varnothing), \bar{a})$. There are several possibilities for A_i. If A_i is $R(\bar{a})$ or not($R(\bar{a})$) for R from S we have a contradiction to the minimality of k.

Otherwise a_i must be from L. If A_i is $R(\bar{a})$ or not($R(\bar{a})$) for R from L then we know that $D \models R(\bar{a}(\bar{a}))$ and that k-1 is greater than zero and $D \models \phi_{R^*}(\bar{a}(\bar{a}))(F^{k-2}(\varnothing))$. Since there is only one sentence $\forall x(\neg R(x) \rightarrow R^*(x))$ in P with R^* in its head we must have

$D \models \neg R(\bar{a}(\bar{a}))$

which is a contradiction.

Q.E.D.

6.3 Summary

Let us just summarise the relationship which we have investigated in this chapter - that between logic programming with negation as failure as a query answering tool and DEF as a query language. Because our results have involved the process QEP instead of one of Clark's original processes, we will have to call on the results of chapter four to give the results of this chapter meaning outside of the constructs of this thesis.

First, we combine result 4.2 and lemma 6.1 to get the following result showing that each LPNAF program implements a DEF query.

THEOREM 6.3

For all signatures **L**, for all signatures **S**, for all n-ary Q from **S**, for all logic programs PROG with symbols from **L**∪**S**, there is an n-ary query q from DEF(**L**) such that

for all versions **R**-qep of NAF, for all finite **L**-structures D, for all n-tuples $\bar{a}$ of elements from D,

a) if progex(D) + PROG succeeds $Q(\bar{a})$ in **R**-qep then $\bar{a} \in q(D)$ and

b) if progex(D) + PROG finitely fails $Q(\bar{a})$ in **R**-qep then $\bar{a} \notin q(D)$.

We note that the program may not get answers on some tuples but when it does they will be answers to q.

Such a concept of implementation is inappropriate when we come to consider the other half of the relationship. Given a DEF query we want to find a program which implements it. This is easy if we allow the program to loop or flounder when it wants to and only require it to get the right results (according to the query) when it does get an answer. We could just implement any query by a program which always flounders.

Obviously this is impractical. Fortunately, our result lemma 6.2 here is much stronger - we find a program which succeeds exactly the right answers.

Unfortunately, there is a problem when we relate this result to actual versions of NAF. Since the program, progex (D) + PROG, which we are using varies with the queried structure D, our result theorem 4.3 only allows us to be sure of the existence of a particular version of NAF able to implement the query which may depend on D. It is left as an open problem whether a general maximal selection rule can be found which is able to implement all progex(D) + PROG for all different D.

Here is the best result we can manage now.

THEOREM 6.4

For all signatures **L**, for all n-ary queries q from DEF(**L**), there is a signature **S**, an n-ary symbol Q from **S** and a logic program PROG with symbols from **L** ∪ **S**, such that
for all finite **L**-structures D, there exists a version **R**-qep of NAF such that
for all $\bar{a}$, n-tuples from D,
progex(D) + PROG succeeds $Q(\bar{a})$ in **R**-qep if and only if $\bar{a} \in q(D)$.

Chapter 7

Nesting

We extended our language DE of definite existential queries to DEF by allowing universal quantifications to appear in the bodies of clauses. This has given us a language representing the expressive power of logic programming with negation as failure. In [ChH85], Chandra and Harel extended their language YE (equivalent to our DE) to the language **C** of clausal queries in which negations of derived relation symbols are allowed to appear in the bodies of clauses (provided that a certain partial ordering is respected). A similar approach forms the basis of the stratified database idea of Apt, Blair and Walker [ABW86]. This might seem to be as good or even better a representation of the behaviour of LPNAF. Certainly the onus is on us to justify introducing the alternative DEF.

In this chapter, we first define **C** and stratified databases and then look at their relationship with NAF. Chandra and Harel proved that **C** is as powerful as FO+LFP and therefore, by a result of ours, **C** is as powerful as DEF, and so by another result of ours **C** is as powerful as LPNAF. We emphasise that we have not seen any other proof that stratified databases are equally expressible as LPNAF.

Chandra and Harel do say that NAF provides an operational justification for **C**, suggesting that a stratified database can be implemented directly as a logic program using NAF. Certainly any stratified database looks like a program.

We show that NAF does not provide an operational justification for **C**. The operational effect of negation in a stratified database is as "not provable" rather than

"finitely fails to be proven". We present an example which shows that even when there is no question of floundering, NAF does not always succeed the not(P)'s for which P is "not provable".

We have said that C is as expressible as LP with NAF. We show that implementing a stratified database using LP with NAF essentially involves a compilation process turning the database into a DEF theory before writing the program. Although the database looks like a program this is not generally the program which implements it!

There is also a problem with the task of finding the C query which a program represents. There are interesting LP with NAF queries involving a program which is not a stratified database.

Finally we note that DEF has a much simpler semantics than C.

Thus we consider that we have ample reason for regarding DEF as the query language which best represents LP with NAF. Stratified databases give access to all the same queries but it is much more credible to regard this as a formalisation of a method of query answering based on taking whole relations through a process of fixed points and complementations.

Despite all this, the idea of nesting queries seems appealing in the context of provability languages - perhaps greater expressive power could be gained by nesting and using provability semantics within each level of the nesting. This leads us to the concept of finite implicit definability - theories which implicitly define a relation provided that we stick to finite structures. We show that there is no point nesting on top of finitely implicitly definable queries because straight provability semantics can be used to express the same query in a straight forward way. Using Immerman's collapse result [Imm] we know that DEF queries are all finitely implicitly definable - so transitive closure is for example. In fact, we show that all polynomial time computable queries are. It is left as

an open question whether finite implicit definability is the same as being in PROV $\cap$ $\neg$PROV $\equiv$ USO $\cap$ $\neg$USO $\equiv$ NP $\cap$ co-NP.

7.1 The Clausal Language

In this section we define the query language CLA which contains all the queries expressible by the clausal theories of [ChH85] - i.e. those theories in C. These theories are constant-free versions of the stratified databases of [ABW86]. CLA queries are not derived from the clausal theories by provability as we have defined it. Thus we must spend some time specifying their different semantics.

Chandra and Harel had defined a class of theories H' which is the same as our class of definite existential theories. Thus, by using basic negations and inequalities in the bodies of otherwise Horn-clauses, they could answer all the DE queries (equivalent in expressibility to their fixed point language YE). They noted, on the other hand, that H' theories could not be used to define the universal quantification query.

Instead of remedying this deficiency, as we do, by adding universal quantification explicitly to get DEF, Chandra and Harel decided to allow the negations of derived relations in the bodies of clauses in a restricted manner. They then had to define a new non-provability semantics for their language as we shall see.

Chandra and Harel describe how to build a theory of their language by composing several DE (i.e. H') theories. We shall, instead, describe the finished product of these operations which now goes under the name of a stratified database.

DEFINITION

If **L** and **S** are disjoint signatures then an extended (L,S)-implication is an L$\cup$S-

sentence of the form $\forall x(\bigwedge_{i=1}^{n} A_i \rightarrow P(\tau))$ where P is from S and for each i=1...n A_i is either u=v, $u \neq v$, R(α) or $\neg$R(α) for R from L or S.

We note that the extended clauses of [ChH85] do not contain negations of derived relations but after composition of several such DE theories as described by Chandra and Harel we end up with extended implications.

DEFINITION

If L and S are disjoint signatures then an (L,S) stratified database is an L$\cup$S-theory consisting entirely of extended (L,S)-implications such that the following recursion limiting condition is satisfied:

there exists a partition $\pi = \{ S_1, \cdots, S_m \}$ of S and a partial ordering $\leqslant$ on π such that

1) if R appears in the body of an implication headed by S then either R is from L or R is from S_j, S is from S_k and $S_j \leqslant S_k$; and

2) if R appears negated in the body of an implication headed by S then either R is from L or R is from S_j, S is from S_k and $S_j < S_k$.

For example, if L contains 2-ary P and S contains 2-ary T and 2-ary R then the following theory is a stratified database:

$\{ \forall xy(P(x,y) \rightarrow T(x,y))$

$\forall xyz(P(x,z) \wedge T(z,y) \rightarrow T(x,y))$

$\forall xy(\neg T(x,y) \rightarrow R(x,y)) \}.$

An ordering is {T} < {R}.

If stratified databases were used as provability theories then they would not be very implementable. Allowing negations of derived relation symbols in the bodies of clauses is equivalent to allowing disjunctions in the head. As pointed out in [GMN84a] the resulting

indefinite deductive databases are much harder to work with.

Instead, the non provability language CLA is defined as follows:

syntax of CLA(L):

CLA(P;R) is a wff of CLA(L) if and only if S is a signature disjoint from L, R is from S and P is an (L,S) stratified database.

semantics of CLA(L):

To specify the meaning of CLA(P;R) on an L-structure D we proceed by induction along the partial order on the partition of S. It does not matter which partial order on which partition we use but for definiteness let us take the minimal order on the maximal partition which is consistent with the recursion limiting property.

Suppose that we have defined whether $D \models CLA(P;S)(\vec{d})$ or not for all $\vec{d}$ from D, for all S from S_i such that $S_i < S_j$. Now let P_j consist of all the clauses with heads from S_j. Say $S_j^{\cup}$ is $\cup \{ S_i \mid S_i < S_j \}$. So P_j is a definite existential $L \cup S_j^{\cup} \cup S_j$-theory. Define an $L \cup S_j^{\cup}$-structure D_j expanding D by $D_j \models S(\vec{d})$ if and only if $D \models CLA(P;S)(\vec{d})$. Then, for every R from S_j we say $D \models CLA(P;R)(\vec{d})$ if and only if $D_j \models PROV(P_j;R)(\vec{d})$.

Using provability in such a stepwise way might not seem to be too radical a change but it enables a later sub-theory to know what is not provable from an earlier sub-theory.

In our previous example of a stratified database we have two sub-theories -

$P_1 = \{ \forall xy(P(x,y) \rightarrow T(x,y)),\ \forall xyz(P(x,z) \wedge T(z,y) \rightarrow T(x,y)) \}$

and $P_2 = \{ \forall xy(\neg T(x,y) \rightarrow R(x,y)) \}$.

If D has a P path from a to b but no path from b to a then $D \models PROV(P_1;T)$ (a,b) but not $D \models PROV(P_1;T)$ (b,a) and thus $D \models CLA(P;R)$ (b,a) but not $D \models CLA(P;R)$ (a,b).

Chandra and Harel give the following theorem (but they call FO+LFP, FP) :

THEOREM 7.1

For all signatures L, CLA(L) ≡ FO+LFP (L).

Using our results relating DEF to both FO+LFP and to LP with NAF we can conclude that CLA captures all the queries implementable by LP with NAF. This result relating stratified databases to logic programming with negation as failure has not, as far as we know, been proven elsewhere.

THEOREM 7.2

For all signatures L, the class of L-queries implementable by tacking a logic program using negation as failure (but not using constant or function symbols) onto a description of the queried L-structure are precisely those queries expressible by using a stratified database with the semantics defined in [ChH85] or [ABW86].

7.2 Negation as failure versus negation as not provable

We have seen that stratified databases capture the expressive power of LP with NAF. Those databases even look like logic programs. Let us consider the relationship in more detail. First, we try implementing a stratified database and then we try expressing in CLA the query which is implemented by a program.

Negation as failure is often associated with "negation as not provable". One way such an association could be formalised is as follows:
if P is a Horn-clause theory from L and R is from L then all the tuples $\bar{d}$ such that R($\bar{d}$)

is not provable from P are exactly all the tuples such that R($\vec{d}$) finitely fails as a query asked of the logic program representing P. This seems to be what Chandra and Harel are claiming [ChH85] when they say that NAF gives rise to the possibility of using the language CLA.

However, the association, as formalised above, does not hold. Consider the following theory.

{ P(a,b)

P(b,a)

$\forall xy(P(x,y)\rightarrow T(x,y))$,

$\forall xyz(P(x,z)\wedge T(z,y)\rightarrow T(x,y))$,

It can be seen that T(a,c) does not follow from the theory. If we ask T(a,c) as a query to the logic program representing the theory then it is also easy to see that the qep will get into a loop.

That difference between "not provable" and "finite failure" is not important to someone who only requires that the consequences of a Horn-clause theory are precisely the successful queries from the pure logic program. However, to implement the "not provable" semantics directly by negation as failure it is required that a literal not provable from a lower sub-program finitely fails so that the fact of its failure is available to a higher sub-program.

For example, suppose that P is the following theory.

{ $\forall xy(P(x,y)\rightarrow T(x,y))$,

$\forall xyz(P(x,z)\wedge T(z,y)\rightarrow T(x,y))$,

$\forall xy(\neg T(x,y)\rightarrow R(x,y))$ }.

If NAF truly implemented "negation as not provable" then the following logic program would implement the query CLA(P;R) -

T(x,y) ← P(x,y)

T(x,y) ← P(x,z) , T(z,y)

R(x,y) ← not(T(x,y)).

But consider the following {P}-structure. D has domain {a,b,c} and P contains only (a,b) and (b,a). Then (a,c) is a correct answer to the query CLA(P;R) asked of D because (a,c) is not an answer to the query CLA(P;T). However if we apply our NAF query evaluation process to prog(D) followed by the supposed implementation of CLA(P;R) then we will have problems succeeding R(a,c). This is because T(a,c) does not finitely fail from the program, as it does not from the relevant sub-program as we saw above.

We can not claim that stratified databases can be implemented directly in logic programs. Now let us try the converse problem expressing in CLA a query implemented by a logic program.

Consider the short program

Odd(x) ← succ(y,x) , not(Odd(y)).

After a description of any finite totally ordered {succ}-structure D this succeeds Odd(d) for all those elements d an odd number of successors from an element which has no predecessor. That is, Odd(d) succeeds if and only if there is an odd positive integer n and a sequence $e_0, \cdots, e_n$ of elements of D such that

1) $D \models \forall x \neg succ(x, e_0)$,

2) $d = e_n$ and

3) for each i= 0 , ... , n-1 , $D \models succ(e_i, e_{i+1})$.

This program is not stratified database and yet NAF (with a left most selection rule) never flounders answering such queries with this program. It can not be claimed that stratified databases are appropriate for directly representing logic programs with NAF.

While a stratified database is not a direct representation of a logic program neither is a DEF theory. Perhaps, there is some simple transformation for turning a logic program

into an SDB akin to the synthesis construction.

However, looking at our negation of the transitive closure example this does not seem likely. Let us suppose that there is a simple transformation. The relation R is the negation of the transitive closure of P. If we could transform this stratified database into an equivalent logic program in a simple manner then the end result could be transformed into an equivalent DEF theory and then into an equivalent $\overline{LFP}$ prescription. What we have ended up doing is providing an alternative method of constructively showing Immerman's collapse result. Since all the transformations are quite simple it would be probable that we would not end up with the very complicated representation of the negation of the ancestor (i.e. transitive closure) query that Immerman gives as an example. So, finding a simple way of implementing a stratified database is essentially the same problem as collapsing LFP operators. This does not seem to be a "simple" procedure : it is certainly not as simple as our implementation for DEF queries.

In summary, we do not think that stratified databases provide a reasonable "logical" representation of logic programs with NAF. They certainly can not be implemented directly and finding a behaviour-preserving transformation is akin to collapsing nestings of LFP operators. There is ample justification for regarding the use of DEF theories as more sensible.

7.3 Nesting

Despite the implementation problems of stratified databases the idea of building provability theories by nesting some within others seems appealing. It would be useful if some queries could be defined in terms of others. In this section we first establish that this is something worth doing but is not trivially achievable. Then we will go on to consider under what circumstances it is useful.

It does not take much searching to find circumstances where defining queries in terms of other queries is desirable. For example, suppose that we are given a set of people and are told who is a parent of whom and who is male. Eventually, we want to compute all the ordered pairs of people such that the first person is an uncle of the second. It seems perfectly sensible to first define grandparents and then define uncles in terms of parents, males and grandparents.

First, let us define what it means for one query to be defined in terms of another. If **L** is a signature, **P** is an m-ary relation symbol not in **L**, **p** is an m-ary **L**-query, **q** is an n-ary **L**-query and r is an n-ary **L**∪{P}-query then we say that q can be *defined* as r *in terms* of p if and only if, whenever any finite **L**-structure D is expanded to an **L**∪{P}-structure E by interpreting P as p(D) then r(E) = q(D).

For example, suppose that **L** contains the 2-ary relation symbols parent and 1-ary male and that grandparent and uncle are other 2-ary relation symbols. Then, on structures with domains of people with parent(a,b) being true if and only if a is a parent of b and male(a) being true if and only if a is male we find that

p= PROV($\{\forall xy(parent(x,y)\wedge parent(y,z)\rightarrow grandparent(x,z)\}$; *grandparent*)

has correct answers exactly those (a,b) such that a is a grandparent of b. If r is the 2-ary **L**∪{*grandparent* , *uncle* }-query

PROV($\{\forall xyu(male(x)\wedge grandparent(u,y)\wedge parent(u,x)\wedge\neg parent(x,y)\rightarrow uncle(x,y))\}$;

uncle)

then it is clear that q can be defined as r in terms of p where q is the 2-ary **L**-query with answers (a,b) such that a is an uncle of b.

It might be objected that this idea is nothing special. We define the "uncle" query in terms of the "grandparent" query just by writing one theory with two sentences -

$\mathbf{P} = \{ \forall xy(parent(x,y) \wedge parent(y,z) \rightarrow grandparent(x,z)),$

$\forall xyu(male(x) \wedge grandparent(u,y) \wedge parent(u,x) \wedge \neg parent(x,u) \rightarrow uncle(x,y)) \}.$

Surely we are doing the same thing every time we define a query from PROV(L) by using a new signature S which contains more than one symbol.

However, that argument contains a basic flaw which is more apparent in other examples. Suppose that we have 0-ary Q in L and two other 0-ary relation symbols P and R. Let p be the L-query PROV($\{ Q \rightarrow P \}$; P) and r the L∪{P}-query,

PROV($\{ \neg P \rightarrow R \}$; R).

It is easy to check that ¬p can be defined as r in terms of p - if an L-structure is expanded to an L∪{P}-structure E by interpreting P as p(D) then E will be a model of P if and only if D⊨Q, but then r(E) is true if and only if E is not a model of P i.e. if and only if ¬p(D) holds. Now, if someone continues to claim that defining queries in terms of other queries is just having big theories relating derived relation symbols to each other then they should claim that ¬p is the L-query PROV($\{ Q \rightarrow P , \neg P \rightarrow R \}$; R). It is easy to see that, on the contrary, the latter is the always false 0-ary L-query.

Thus, the difference between nesting queries and joining theories is a generalisation of the difference between the stratified database semantics for a theory (used in a query) and the provability semantics. In the example above we note that CLA($\{ Q \rightarrow P, \neg P \rightarrow R \}$; R) is actually ¬p.

In the stratified database situation we have seen that the "nesting" semantics does not give us any increased expressive power over DEF. In this section we will consider to what extent nesting of provability queries in general gives us access to other non-provability theories.

It is not hard to see that there probably are ways of nesting provability queries to escape from that class of queries. We have seen that PROV(L) contains exactly those 0-

ary L-queries which are in co-NP i.e. which have negations which are non-deterministically polynomial time computable. It is an open problem whether NP = co-NP or not (see [GaJ79]). If NP ≠ co-NP then (because of the symmetry of the situation with respect to negation) there will be L-queries in co-NP but not in NP. Such queries will be in PROV(L) but their negations will not. If we take one such p then it is clear that ¬p can be defined as r = PROV($\{\neg P \rightarrow R\}$; R) in terms of p , i.e. nesting provability r on top of provability p gets us out of PROV(L).

The main result of this section will be showing a partial converse of that observation. We will find a sufficient condition on p for there to be no point nesting on p - we could easily express the result in PROV(L) anyway.

For any signature L, non-negative integer n, n-ary L-query p and n-ary symbol $P \notin L$ we say that p is *collapsible* if and only if for every query r from PROV(L∪{P}), if q can be defined as r in terms of p then q ∈ PROV(L).

So the collapsible queries are those which can not be stepping stones to get out of PROV. We already know that if any queries are not in NP then they are not collapsible. It is easy to show that if a query is not in co-NP then it is not collapsible either.

RESULT 7.3

p collapsible implies that p is in NP ∩ co-NP.

We try for a converse.

In our p= PROV($\{Q \rightarrow P\}$; P) and r = PROV($\{\neg p \rightarrow R\}$; R) example we noted that PROV($\{Q \rightarrow P, \neg P \rightarrow R\}$; R) does not express the query q which can be defined as r in terms of p. Our investigations in the last section should suggest that we could try expressing ¬p as a provability theory. Finding that ¬p = PROV($\{\neg Q \rightarrow P'\}$; P') it is then

not surprising that q = PROV($\{Q \rightarrow P, \neg Q \rightarrow P', P' \rightarrow R\}$; R).

This can be generalised into a possible method of collapsing nestings. Suppose that **L**, **S** and **S'** are disjoint signatures, m-ary P is from **S**, m-ary P' is from **S'** and ¬PROV(P;P) = PROV(P';P'). If PROV(R;R) is an implicational **L**∪{P}-query and q can be defined as PROV(R;R) in terms of PROV(P;P) then we could try expressing q by first changing **R** into **R'** by changing each appearance of ¬P in a body into P' and then seeing whether q is PROV(**P**∪**P'**∪**R'**; R).

DEFINITION

Let us say that (**P**,**P'**) is (P,P')-*nestable* if

1) PROV(P';P') = ¬ PROV(P;P); and

2) for every implicational (**L**∪{P} , **T**)-theory **P**, for all R from **T** if q can be expressed as PROV(R;R) in terms of PROV(P;P) then q = PROV(**P**∪**P'**∪**R'**;R).

DEFINITION

We will say that an n-ary **L**-query p is *easily collapsible* if there are **P** , **P'**, P and P' such that p= PROV(P;P) and (**P**,**P'**) is (P,P')-nestable.

Since all PROV(**L**∪{P}) queries can be expressed as implicational PROV(**L**∪{P}) queries it is not too hard to see that easily collapsible implies collapsible.

Now we recall that we hope to show that every **L**-query which is also from NP∩co-NP is collapsible. Things are looking good because, if p is in NP∩co-NP then both p and ¬p will be provability queries - so there is **P**, **P'** , P and P' such that p= PROV(P;P) and ¬p = PROV(P';P').

Unfortunately, we can not expect just any such (**P**,**P'**) to be (P,P')-nestable. Consider the following example. Suppose that **L** is empty and P is a 1-ary relation symbol. Let p be the always full 1-ary **L**-query which is clearly in NP∩co-NP. As they should be, p and ¬p are able to be expressed as PROV(**L**) queries. For example,

p = PROV($\{\forall x P(x)\}$; P)

and ¬p = PROV($\{\forall x y(x \neq y \rightarrow (P'(x) \vee P'(y)))\}$;P').

Its clear that PROV($\forall x P(x)\}$; P) is the full 1-ary **L**-query. The expression which we are claiming represents ¬p can be seen to be the always empty 1-ary **L**-query, as , given any finite **L**-structure (i.e. set) D, and any d from D, we can expand D to a model E of the theory by making P'(e) true of every e except d - i.e. it is not so that

$D \models$ PROV($\{\forall x y(x \neq y \rightarrow (P'(x) \vee P'(y)))\}$;P') (d).

We are not claiming that this expression is the most "simple" for ¬p. (**P**,**P'**) is not (P,P')-nestable. Take R to be a 0-ary relation symbol and **R** to be $\{\forall x(P(x) \wedge \neg P(x) \rightarrow R)\}$. Clearly r = PROV(**R**;R) is the always false 0-ary **L**∪{P}-query and so the always false 0-ary **L**-query q can be expressed as PROV(**R**;R) in terms of p. Given any **L**-structure D , if we interpret P as the full 1-ary relation (i.e. p(D)) to get the expansion E of D then r(E) and q(D) are both falsity. The result of nesting (**P**,**P'**)∪**R'**

is $\{\forall x P(x)$, $\forall x y(x \neq y \rightarrow (P'(x) \vee P'(y)))$, $\forall x(P(x) \wedge P'(x) \rightarrow R)\}$. But any expansion to a model of this of a set with more than one element must have R true. So PROV(**P**∪**P'**∪**R'**;R) is not q.

While this (**P**,**P'**) is not (P,P')-nestable it is clear that p, here, is easily collapsible - if we express ¬p as PROV(∅;P') instead then (**P**,**P'**) is (P,P')-nestable. Now let us try to find general conditions under which we can find the appropriate (**P**,**P'**) ;↓ there is one.

7.4 Finite Implicit Definability and Minimal Models

We have narrowed down our investigations of nesting to a search for easily collapsible queries. It turns out that this is related to the search for finitely implicitly definable queries and to the search for minimisable queries. Let us define these two new concepts and then give a result relating the three properties.

We shall say that an L∪S-theory Q *finitely implicitly defines* (fids) an n-ary L-query p as n-ary P from S if and only if

1) PROV(Q;P) = p and

2) for all finite L-structures D, there is an expansion of D to a model of Q and any two such expansions agree on their interpretations of P.

We shall say that p is finitely implicitly definable (fid) if there is such a theory Q.

So if Q fids p as P then Q not only defines p in the provability sense but also, at least on finite structures, Q defines p as P in the usual first-order sense - any two models which agree on their L-reducts agree on their interpretations of P.

It is not hard to show that any FO(L) query is fid. Say that p is an n-ary L-query defined by the FO(L) formula $\phi(\bar{x})$. If we choose n-ary Q not in L, and let Q be $\{\forall \bar{x}(\phi(\bar{x}) \longleftrightarrow Q(\bar{x}))\}$ then Q fids q as Q. First we show that p is PROV(Q;Q). If D is a finite L-structure, $\bar{d}$ is from D and D ⊨ PROV(Q;Q) $(\bar{d})$ then let E be the expansion of D to the L∪{Q}-structure got by interpreting Q to contain all $\bar{a}$ such that D⊨$\phi(\bar{a})$. Then E⊨Q so E⊨$Q(\bar{d})$. By our construction D⊨$\phi(\bar{d})$ so $\bar{d} \in$ q(D). If $\bar{e} \in$ q(D) then D⊨$\phi(\bar{e})$ and if we take any expansion E of D to a model of Q we will have E⊨$\phi(\bar{e})$ so E⊨$Q(\bar{e})$.
Hence D ⊨ PROV(Q;Q) $(\bar{e})$. Any finite L-structure D has an expansion to the model of Q got by interpreting Q as ϕ on D and if any two expansions of D are both models of Q then Q will be interpreted as ϕ on D in both.

If finite implicit definability was the same as (not necessarily finite) implicit definability then the first order queries would be the only fid queries (using Beth's theorem). However, we shall see in the next section that there are more fid queries.

The other concept which we will see is relevant here is the idea of a theory having minimal models. We shall look at those provability queries PROV(P;P) such that P has minimal models with respect to their interpretations of P. In chapter two we considered theories with minimal expansions with respect to all of their relation symbols. If L is a signature and p an n-ary L-query then we say that p is *minimisable* if and only if there is a signature S disjoint from L, an n-ary relation symbol P in S and an LUS-theory P such that

1) PROV(P;P)⊨P and

2) for every finite L-structure D, there is an expansion E of D to a model of P such that P is interpreted as p(D) in E.

Since p = PROV(P;P) in this case, any expansion of D to a model of P will include those tuples in p(D) in its interpretation of P. Thus E is a minimal model with respect to the interpretation of P under the set inclusion ordering on extensions of relations.

It is well known that Horn-clause theories have minimal models (minimal on all interpretations simultaneously) so all DATALOG queries are minimisable. We have generalised this result to DEF queries in chapter two. In the next section we will go further. For now let us relate our three properties.

LEMMA 7.4

The following are equivalent :

1) p is easily collapsible

2) p and ¬p are minimisable

3) p is fid.

proof:

part one (easily collapsible implies minimisable and negation minimisable)

Suppose that the n-ary L-query p is easily collapsible. So there are **P**, **P'**, P and P' such that p = PROV(**P**;P) and (**P**,**P'**) is (P,P')-nestable. Let D be any finite L-structure.

If r = PROV($\{\forall x(P(x)\wedge\neg P(x)\rightarrow R)\}$; R) then r is the always false 0-ary L$\cup${P}-query. Since (**P**,**P'**) is (P,P')-nestable and the always false 0-ary L-query q can be expressed as r in terms of p then q = PROV(**P**$\cup$**P'**$\cup$**R'** ; R) where **R'** is $\{\forall x(P(x)\wedge P'(x)\rightarrow R)\}$.

q is the always false query so we do not have D $\models$ PROV(**P**$\cup$**P'**$\cup$**R'**;R). Hence, there exists an expansion E of D to a model of **P**$\cup$**P'**$\cup$**R'** such that E is not a model of R.

It is not hard to show that the interpretation of P in E is p(D).

If $\alpha \in$ p(D) then D $\models$ PROV(**P**;P) (α). E$\models$**P** so E$\models$P(α).

If E$\models$P(α), since E$\models\forall x(P(x)\wedge P'(x)\rightarrow R)$ but E is not a model of R then E is not a model of P'(α). E$\models$**P'** so we do not have D $\models$ PROV(**P**;P') (α). Thus α is not in $\neg$p(D) so α is in p(D).

Thus p is minimisable. Since (**P**,**P'**) being (P,P')-nestable implies that $\neg$p = PROV(**P'**;P') we can similarly show that $\neg$p is minimisable too.

end of proof of part one

part two (minimisable and negation minimisable implies fid)

Suppose that p = PROV(**P**;P), $\neg$p = PROV(**P'**;P') and that any finite L-structure can be expanded both to a model of **P** minimal in its interpretation of P and to a model of P'

minimal in its interpretation of P'. Without loss of generality we may suppose that P and P' share no symbols in common apart from those in L.

Let $Q = P \cup P' \cup \{ \forall x(P(x) \longleftrightarrow \neg P'(x))\}$. We will show that Q fids p as P.

First we show that each finite L-structure D has an expansion to a model of Q. We know that we can expand D to a model E of P on which P is interpreted as p(D) and that we can expand it to a model F of P' on which P' is interpreted as $\neg$p(D).

We combine E and F to get an expansion G of D as follows. The symbols from P which are not in L are interpreted in G as they are in E while the symbols from P' which are not in L are interpreted in G as they are in F. Thus G is a model of $P \cup P'$. It is easy to see that G is a model of $\forall x(P(x) \longleftrightarrow \neg P'(x))$ because P on G is p(D) and P' is $\neg$p(D). So $G \models Q$.

Now we can show that PROV(Q;P) = p. If $\bar{a} \in p(D)$ and $E \models Q$ is an expansion of D then $E \models P$ and $D \models$ PROV(P;P) $(\bar{a})$ so $E \models P(\bar{a})$. Conversely, if $\bar{a} \notin p(D)$ then $\bar{a} \in \neg p(D)$ so, if G is the expansion of D to a model of Q as above, then $G \models P'(\bar{a})$. Thus G is not a model of $P(\bar{a})$ so it is not so that $D \models$ PROV(Q;P) $(\bar{a})$

Finally we show that expansions of a finite L-structure D to models E and F of Q agree on their interpretations for P. If $E \models P(\bar{a})$ then E is not a model of $P'(\bar{a})$. Thus we do not have $D \models$ PROV(P';P') $(\bar{a})$. So $\bar{a} \notin \neg p(D)$ and hence $\bar{a} \in p(D)$. So $D \models$ PROV(P;P) $(\bar{a})$ and we have $F \models P(\bar{a})$ as required.

end of part two of proof

part three (fid implies easily collapsible)

Suppose that n-ary p is fid - say , Q fids p as P. Thus PROV(Q;P) = p. Let Q' be $Q \cup \{\forall x(P(x) \longleftrightarrow \neg P'(x)\}$. We will show that (Q,Q') is (P,P')-nestable.

First, we must show that PROV(Q';P') = ¬ p. Let D be any finite L-structure and $\bar{a}$ from D.

If D ⊨ PROV(Q';P') $(\bar{a})$ then we can find an expansion E of D which is a model of Q. We expand E to F by defining P' to be the complement of P. So F is a model of Q' expanding D and hence F⊨P'$(\bar{a})$. Thus, F⊨¬P$(\bar{a})$, we can not have D ⊨ PROV(Q;P) $(\bar{a})$, $\bar{a} \notin$ p(D), and so $\bar{a} \in$ ¬p(D) as required.

Conversely, if $\bar{a} \in$ ¬p(D) then we do not have D ⊨ PROV(Q;P) $(\bar{a})$, so there is an expansion E of D to a model of Q such that E is not a model of P$(\bar{a})$. Now suppose that G is any expansion of D to a model of Q'. Then G is an expansion of D to a model of Q so G and E agree on P. Hence G is not a model of P$(\bar{a})$ but since it is a model of $\forall x(P(x) \longleftrightarrow \neg P'(x))$ we have G⊨P'$(\bar{a})$. So D ⊨ PROV(Q';P') $(\bar{a})$.

Now suppose that R is an implicational theory and that q can be expressed as PROV(R;R) in terms of p. We will show that q = PROV(Q∪Q'∪R'; R). Suppose that we have a finite L-structure D and $\bar{a}$ from D.

First we show that $\bar{a} \in$ q(D) implies that D ⊨ PROV(Q∪Q'∪R';R) $(\bar{a})$. If $\bar{a} \in$ q(D) then we take any expansion G of D to a model of Q∪Q'∪R' and, since G⊨$\forall x(P(x) \longleftrightarrow \neg P'(x))$ we will have G⊨R.

Now if F is the reduct of G to L∪{P} we find that the interpretation of P in F is p(D). For if F⊨P$(\bar{c})$ then G⊨P$(\bar{c})$ so G is not a model of P'$(\bar{c})$, we do not have D ⊨ PROV(Q';P') $(\bar{c})$, $\bar{c} \notin$ ¬p(D) and so $\bar{c} \in$ p(D). On the other hand, if $\bar{c} \in$ p(D) then D ⊨ PROV(Q;P) $(\bar{c})$, G⊨P$(\bar{c})$ and so F⊨P$(\bar{c})$.

Since q can be expressed as PROV(R;R) in terms of p, q on D and PROV(R;R) on F have the same answers. Thus D ⊨ PROV(R;R) $(\bar{a})$. But G is an expansion of D to a model of R so G⊨R$(\bar{a})$. This goes for any such expansion G so D ⊨ PROV(Q∪Q'∪R';R)

(σ).

Next we show that if $D \models PROV(Q \cup Q' \cup R';R)$ (σ) then $\sigma \in q(D)$. Suppose that $D \models PROV(Q \cup Q' \cup R';R)$ (σ). Since Q fids p as P there is an expansion H of D to a model of Q.

We note that the interpretation of P in H is p(D). If $\sigma \in p(D)$ then $D \models PROV(Q;P)$ (σ) so $H \models P(\sigma)$. If $H \models P(\sigma)$ then, since all expansions of D to models of Q agree on their interpretations for P , $D \models PROV(Q;P)$ (σ) and so $\sigma \in p(D)$.

Now let F be the expansion of D to an $L \cup \{P\}$-structure got by interpreting P as p(D). If G is any expansion of F to a model of R then we form a structure I from the signature containing L, {P,P'}, all the relation symbols Q and all those from R. The interpretations for symbols from L are as on D, for P as on F (and H), for P' the complement of P, for the others from Q as on H and others from R as on G.

It is clear then, that I is a model of $Q \cup Q' \cup R'$ so $I \models R(\sigma)$. But R has interpretation in I as in G so $G \models R(\sigma)$. This proves that $D \models PROV(R;R)$ (σ) . Now, since q can be defined as PROV(R;R) in terms of p we have $\sigma \in q(D)$ as required.

This shows that (Q,Q') is (P,P')-nestable.

Q.E.D.

7.5 Polynomial Time Computable Queries

We are now able to prove a result which shows that nesting of queries is not such a useful technique. If a query is computable in a reasonable amount of time we will show that there is no point nesting on it.

First let us recall some properties of DEF.

LEMMA 7.5

$p \in DEF(L)$ implies that p is minimisable

proof:

Suppose that $p = PROV(P;P)$ where P is a definite (L,S)-theory. Lemma 2.6 showed that P is minimisable - i.e. for each D there is an expansion of D to a model of P which is minimal with respect to each of the symbols in S. In particular it is for P.

Q.E.D. (lemma 7.5)

LEMMA 7.6

DEF queries are easily collapsible

proof:

We have just proved that DEF queries are minimisable. Using our result showing the equivalence of DEF and FO+LFP and the fact that FO+LFP is closed under negations we can also deduce that the negations of DEF queries are also minimisable. By lemma 7.4 of the last section any DEF query is thus easily collapsible.

end of proof

The following theorem from [Imm82] shows that on totally ordered structures, polynomial time computable queries are precisely the FO+LFP queries with one level of nesting (which we know means all the FO+LFP queries).

THEOREM 7.7

If **L** is a signature, $\leqslant$ a 2-ary relation symbol not in **L** and q an $\mathbf{L}\cup\{\leqslant\}$-query then q is computable in polynomial time on all $\mathbf{L}\cup\{\leqslant\}$-structures totally ordered by $\leqslant$ if and only if
there is a formula $\phi(x) = \phi$ from $FO(\mathbf{L}\cup\{P\}\cup\{\leqslant\})$ and a tuple x of variable symbols such that q and $LFP_{P,x}(\phi)(x)$ agree on all finite **L**-structures with $\leqslant$ interpreted as a total order.

Using this theorem we can generalise the last lemma.

LEMMA 7.8

p is polynomial time computable implies that p is minimisable

proof:

Suppose that **L** is a signature, p is an n-ary **L**-query and that p is polynomial time computable.

We choose a 2-ary relation symbol $\leqslant$ not in **L** and define an n-ary $\mathbf{L}\cup\{\leqslant\}$-query q by $q(G) = p(G|_{\mathbf{L}})$ on all finite $\mathbf{L}\cup\leqslant$-structures G. Thus q is polynomial time computable on all $\mathbf{L}\cup\leqslant$-structures so it is polynomial time computable on all $\mathbf{L}\cup\leqslant$-structures which are totally ordered by $\leqslant$.

By Immerman's result q is equivalent to a $FO+LFP(\mathbf{L}\cup\leqslant)$ query on those structures and by our theorem 3.19 this in turn is equivalent (on all structures) to a DEF query. Suppose that PROV(P;Q) is one such DEF query.

Now let $TO_{\leqslant}$ be the following theory which says that $\leqslant$ is a total order.

$TO_{\leqslant} = \{ \quad \forall xy(x \leqslant y \vee y \leqslant x),$

$\forall x(x \leqslant x)$,

$\forall xy(x \leqslant y \wedge y \leqslant x \rightarrow x = y)$,

$\forall xyz(x \leqslant y \wedge y \leqslant z \rightarrow x \leqslant z)$ }.

We put **Q** equal to $TO_{<} \cup \mathbf{P}$.

We show that p is minimisable by showing that 1) PROV(**Q**;Q) = p and 2) that every finite **L**-structure D can be expanded to a model of **Q** on which Q is interpreted as p(D).

1.1) $D \models \mathrm{PROV}(\mathbf{Q};Q)\ (\bar{a})$ implies $\bar{a} \in p(D)$

If $D \models \mathrm{PROV}(\mathbf{Q};Q)\ (\bar{a})$ then let the $\mathbf{L} \cup \leqslant$-structure G be got from D by totally ordering its domain in some way. If H is any expansion of G to a model of **P** then H is an expansion of D to a model of **Q** so $H \models Q(\bar{a})$. Thus $G \models \mathrm{PROV}(\mathbf{P};Q)\ (\bar{a})$ and since PROV(**P**;Q) and q are equivalent on totally ordered structures such as D we have $\bar{a} \in q(G)$. But by definition of q we have $\bar{a} \in p(D)$ as required.

1.2) $\bar{a} \in p(D)$ implies $D \models \mathrm{PROV}(\mathbf{Q};Q)\ (\bar{a})$

Suppose that $\bar{a} \in p(D)$. If H is any expansion of D to a model of **Q** we put $G = H|_{\mathbf{L}} \cup \leqslant$ so that G is totally ordered by $\leqslant$. But $q(G) = p(D)$ so $\bar{a} \in q(G)$.
Thus $G \models \mathrm{PROV}(\mathbf{P};Q)\ (\bar{a})$ and since H is an expansion of G to a model of **P** we have $H \models Q(\bar{a})$. Thus $D \models \mathrm{PROV}(\mathbf{Q};Q)\ (\bar{a})$ as required.

2) Every finite **L**-structure D can be expanded to $H \models \mathbf{Q}$ such that Q is interpreted as p(D) on H.

We expand D to an $\mathbf{L} \cup \leqslant$-structure G by totally ordering its domain in some way.

Since **P** is a definite theory, PROV(**P**;Q) is minimisable so that G can be expanded to a model H of **P** such that Q is interpreted to contain exactly those $\bar{a}$ such that
$G \models \mathrm{PROV}(\mathbf{P};Q)\ (\bar{a})$.

As G is totally ordered H is a model of Q. We need only show that Q is interpreted as p(D) on H.

If H⊨Q(σ) then D ⊨ PROV(P;Q) (σ) so $\sigma \in$ q(G) and so $\sigma \in$ p(D).

If $\sigma \in$ p(D) then D ⊨ PROV(Q;Q) (σ) so any I which is a model of P expanding G is also a model of Q expanding D and so satisfies Q(σ). Hence D ⊨ PROV(P;Q) (σ) so H⊨q(σ).

This completes the proof of the theorem.

COROLLARY 7.9

polynomial time computable implies easily collapsible

proof:

This follows easily from lemma 7.4 since negations of polynomial time computable queries are too.

Q.E.D. (corollary 7.9)

So there really is no point in nesting provability queries unless you are using very time consuming ones.

We may have reached the desired converse of result 7.3 but it will have to wait until it is shown whether NP ∩ co-NP is just the polynomial time computable queries or not before we know for certain.

Chapter 8

Improved Logic Programming

We have demonstrated a very close relationship between the results of answering queries using logic programming and of expressing queries in the logic language DEF. On a more general level we have related logic programming to certain logic theories using the synthesis construction. In the context of supposedly declarative programming languages it seems desirable if we could carry these relationships a step further and see if we can use the mechanics of logic programming directly on the logical theories to produce a truly declarative logic programming language. In this chapter, we do just this and introduce an idea for a logic programming language extended to handle universal quantifications and inequalities but free from the non-declarative features of NAF. However, we will only present the language in a preliminary function-free version.

8.1 QUILOG

In this section we will specify what we want from a declarative programming language powerful enough to be a rival to LP with NAF. From our results relating NAF programs to their syntheses, we know that we must add inequalities and universal quantifications to the repertoire of logic programming. We define a language QUILOG - Quantifications (including Universal) and Inequalities in LOGic programming. It will be designed to be declarative, to be based on the usual query evaluation process for logic programming and to be able to handle programs representing the syntheses of NAF

programs. This last requirement will ensure that QUILOG can answer all the problems that LP with NAF does.

We will have a program as a list of clauses as usual but now the clauses can have more in their bodies. We will call the formulas listed in the bodies of the clauses, *con juncts*.

If **L** is a signature and **C** a finite set of constant symbols then a QUILOG *program* from **L**∪**C** is a finite sequence of QUILOG clauses. A QUILOG clause is a string of the form $R(\tau) \leftarrow A_1, \cdots, A_q$ where each A_i is either

1) $S(\sigma)$ for k-ary S from **L** and σ a k-tuple of **C**-terms,

2) $v \neq w$ for v and w **C**-terms or

3) $\forall y\ S(\sigma)$ for S k-ary from **L** and σ a k-tuple of **C**-terms.

In order to spare us from being too declarative and having to list that each of the constant symbols is different from each of the others, we shall suppose that the semantics of the language takes care of those sentences. We say that PROG *represents* [PROG] $\cup INEQ(\mathbf{C})$

where $INEQ(\mathbf{C}) = \{\ a \neq b \mid$ a,b different elements of **C** $\}$,

[PROG] = { [c] | c is a clause from PROG }

and for each clause c, $R(\tau) \leftarrow A_1, \cdots, A_q$, from PROG with variables τ in some order, [c] is $\forall \tau (\bigwedge_{i=1}^{q} A_i \rightarrow R(\tau))$.

It is clear what we want from the query evaluation process. If R is from **L** and α a tuple of constant symbols from **C** and variable symbols then, as before, we call $R(\alpha)$ a positive literal. The substitution θ of the variables τ in α is a correct answer to $R(\alpha)$ asked of PROG if and only if [PROG] $\cup INEQ(\mathbf{C}) \vdash \forall \tau\ R(\alpha\theta)$. The correct answer is "no" if and only if it can not be deduced that $\exists \tau\ R(\alpha)$ from [PROG] $\cup INEQ(\mathbf{C})$. If there are no variables in α and the empty substitution is a correct answer then so is "yes".

Before going on to develop such a process in the next section, let us look at an example. From the program

popular(y) ← $\forall x$ likes(x,y)

likes(Mary , Mary) ←

likes(John , Mary) ←

likes(x , Mary) ← $x \neq$Mary , $x \neq$John

we want popular(Mary) to succeed. If the third line was not there then we would want it to fail.

8.2 The Query Evaluation Process

Now let us develop a query evaluation process for QUILOG. As usual we will have goals, selection rules and most general unifiers.

A QUILOG goal is a string of the form $\leftarrow A_1, \cdots, A_t$ where each A_i is of the form $S(\sigma)$, $v \neq w$ or $\forall y\ S(\sigma)$. Each A_i, whether a literal , an inequality or a universal quantification is called a *con junct* of the goal.

A *selection rule* is a map which picks one conjunct from each non-empty goal.

If G is a QUILOG goal then a substitution of its free variables is called a substitution of it or of its variables. If θ is a substitution of the free variables in a conjunct L (plus maybe others) then Lθ is a predetermined formula such that

1) it is Lθ as usual if L is an inequality or a literal, and

2) if L is a universal quantification, say $\forall y S(\alpha)$ then there is a variable symbol z which is not in the range of θ such that if x are the variables other than y in α then Lθ is $\forall z[S(\alpha)\ (x \backslash x\theta, y \backslash z)\ ]$.

If θ is a substitution of G = $\leftarrow L_1, \cdots, L_p$, then G$\theta$ is $\leftarrow L_1\theta, \cdots, L_p\theta$. For example, θ

taking x to x1 and y to y1 is a substitution of G = $\leftarrow \forall y S(x,y), R(y)$ and $G\theta$ might be $\leftarrow \forall z S(x1,z), R(y1)$.

We will only need to talk about mgus of literals (not other types of conjuncts) from QUILOG goals and heads from clauses. This idea is the straight forward generalisation of the concept from the pure logic programming case.

Now, if a literal is selected then we know what to do - we proceed as in SLD-resolution. Let us consider first what to do if, instead, an inequality is selected.

If $a \neq a$ is selected then clearly we want the goal to fail.

If $a \neq b$ is selected for different a and b then the conjunct follows from [PROG] $\cup$ INEQ(C), so we can get a new goal by just removing this literal.

Because a non-ground $x \neq a$ or $x \neq y$ can turn into either of the above when we ground more variables as we move down the path, it is best if we leave such inequalities until they are ground. However, this leaves us with the possibility of ending up with a goal full of non-ground inequalities. Suppose that we started with the goal G, ended up with a list d of non-ground inequalities and went through a list of unifiers composing to θ. In the next section, we will see that the QUILOG process, which we define, will, in this case, guarantee that $\forall(\wedge d \rightarrow \wedge G\theta)$ is provable from [PROG] $\cup$INEQ(C). As we will see, this is a useful thing to know when we come to have to deal with universal quantifications. We stop the process under these conditions and say that d implies $G\theta$.

If d implies $G\theta$ and a term $t \neq t$ appears in d then $\wedge$d can never be true so the result is not much use. A different problem arises if there is a variable symbol in d which is not in $G\theta$. Suppose, for example, that $x \neq a$ implies R(y)θ when θ takes y to a. So we know that $\forall x\ (x \neq a \rightarrow R(a))$. If there is another constant other than a appearing in the program then we would know that R(a) follows from PROG $\cup$ INEQ(C). Otherwise, if a is the

only constant in PROG then we would have to know whether C contains other constants to know whether this is a valid deduction or not. In fact, the author is confident that such questions need never be asked when asking, in QUILOG, about deductions from the program representing the synthesis of a NAF program (this confidence is summed up in Conjecture 8.2 in the next section). We will let QUILOG flounder in such circumstances - i.e. when variables appear in d which are not in $G\theta$ - and take this as an indeterminate answer - a don't know.

If d implies $S(\alpha)\theta$ and d contains neither a conjunct of the form $t \neq t$ nor any variables not in $\alpha\theta$, then we say that $S(\alpha)$ succeeds with substitution θ under conditions d.

The important aspect of the query evaluation process is its method of handling universal quantifications. Suppose that we have selected $\forall y S(\alpha)$ from the last goal . We leave this path for the moment and start another one for $S(\alpha)$. The definition of the results of the process will thus have to be done inductively at different levels as in the NAF case. If $S(\alpha)$ finitely fails then we fail our original goal, if it loops then our goal is equally indeterminate but if $S(\alpha)$ succeeds then we have several possibilities. Let us suppose that $S(\alpha)$ succeeds with substitution θ under conditions d.

First let us just get clear what we want from the process. If the path so far is $(G_0, \cdots, G_k)$ with $(\theta_0, \cdots, \theta_{k-1})$ then, as in the pure logic programming case we would like to have ensured that $\forall(\wedge G_k \rightarrow \wedge G_0\theta_0 \cdots \theta_{k-1})$ follows from [PROG] $\cup$ INEQ(C). If we choose a new goal G_{k+1} and a new substitution θ_k such that $\forall(\wedge G_{k+1} \rightarrow \wedge G_k\theta_k)$ follows then we would have the desired result that $\forall(\wedge G_{k+1} \rightarrow \wedge G_0\theta_0 \cdots \theta_k)$ follows.

If G_k is $\leftarrow L_1, \cdots, L_p$ and L_j is $\forall y S(\alpha)$ then, in the usual logic programming style, we want to replace L_j from G_k by a list C of other literals, and find a substitution θ_k so that the new goal G_{k+1} which is $\leftarrow L_1\theta_k, \cdots, L_{j-1}\theta_k, C, L_{j+1}\theta_k, \cdots, L_p\theta_k$, satisfies the

condition above. This will happen if we make sure that $\forall(\wedge C \rightarrow \forall y S(\bar{x})\theta_k)$ (is true in any model of [PROG] ∪ INEQ(C)).

A little bit of work might need to be done here. If, by considering $S(\bar{x})$, we find a substitution ϕ of the variables in $\bar{x}$ such that $\forall(\wedge C \rightarrow \forall y S(\bar{x})\phi)$, we will have to generalise it to a substitution θ_k of all the variables in G_k. As usual we want to gain maximum benefit by using the most general such generalisation. If θ is a substitution of G, ϕ a substitution of some of the variables in G then θ is an *mgg* (*most general generalisation*) of ϕ to G if and only if

1) ϕ is the restriction of θ to its domain and

2) if ϕ is the restriction of substitution θ' to its domain then there is a substitution α of $G\theta$ such that $G\theta\alpha = G\theta'$.

Its not too hard to show that such a substitution exists. We suppose that there is one predetermined mgg called the mgg.

Now let us return to the situation in which, in order to handle the selection of $\forall y S(\bar{x})$ we have gone away and found that $S(\bar{x})$ succeeds with substitution θ under conditions d. So we have ensured that $\forall(\wedge d \rightarrow S(\bar{x})\theta)$.

If y does not appear in $\bar{x}$ then we also know that $\forall(\wedge d \rightarrow \forall y S(\bar{x})\theta)$ and we can put C = d and θ_k as the mgg of θ to G_k to get our next goal.

Now suppose that y does appear in $\bar{x}$.

If $y\theta$ is a constant then we have not learnt much about $\forall y S(\bar{x})$. For example, we might be interested in $\forall y S(x,y)$ and find that $S(x,y)$ succeeds with y being taken to a. Thus we know that S(x , a) is true. If we were to find that $\forall S(x,y)$ is true then it means that we must also be able to show that $\forall y(y \neq a \rightarrow S(x,y))$. This is the crucial observation regarding the qep of QUILOG. When trying to prove $\forall y S(x,y)$, we ignore discoveries such as that of S(x , a) being true and only come back to them if we find that

S(x,y) succeeds under some condition making y different from a constant. In the general process we ignore the result that $S(a)$ succeeds if $y\theta$ is a constant and ask for an alternative path for $S(a)$.

For similar reasons we ignore the result if $y\theta$ is equal to $x\theta$ for a different variable symbol x from a.

So suppose that we find a result in which $y\theta$ is nether a constant nor equal to $x\theta$ for another x. To continue our path we first divide d up into two sets of conditions - those relating $y\theta$ and those not. It is easiest to look at an example now, define the generalised process below and then, in the next section, give a formal proof of the sensibility of its behaviour.

Suppose that $\forall yS(x,y)$ is the selected conjunct and that S(x,y) succeeds under θ taking x to x and y to y , under conditions $x \neq a, y \neq x, y \neq c$. Thus we know that $\forall xy(x \neq a \wedge y \neq x \wedge y \neq c \rightarrow S(x,y))$.
In fact, if e does not equal a and S (e , e) and S(e , c) are true then we will know that $\forall yS(e,y)$ is true. This allows us to have the next goal in the path got by replacing $\forall yS(x,y)$ by $x \neq a$, S(x , x) , S(x , c).

Now let us see how such an example can be generalised.

If G and H are goals, θ a substitution of the variables in G and m a non-negative integer then we say that H is *got* from G *vla* θ at level m if and only if
1) G is $L_1, \cdots, L_p$;
2) L_j is selected from G;
3) either
3a) L_j is $a \neq b$ for two different constant symbols a and b, and H is G without L_j and θ is the identity

3b) L_j is atomic, there is a clause M $\leftarrow A_1, \cdots, A_q$ such that M unifies with L_j with mgus θ , ϕ of L_j from G and M from the clause and H is $\leftarrow L_1\theta, \cdots, L_{j-1}\theta, A_1\phi, \cdots, A_q\phi, L_{j+1}\theta, \cdots, L_p\theta$, or

3c) L_j is $\forall y\ S(\alpha)$ and

3c1) m>0,

3c2) $S(\alpha)$ succeeds at level m-1 with substitution ϕ under conditions d,

3c3) (as defined below) with $S(\alpha)$ succeeding with ϕ under d at level m-1

to show $[\forall y S(\alpha)]\phi$, it remains to show $S(\alpha\phi_1), \cdots, S(\alpha\phi_k)$

and conditions e,

3c4) θ is the mgg of ϕ to G and

3c5) H is $\leftarrow (L_1\theta), \ldots, (L_{j-1}\theta), e, S(\alpha\phi_1), \ldots, S(\alpha\phi_k), (L_{j+1}\theta), \ldots, (L_p\theta)$.

With $S(\alpha)$ *succeeding* with ϕ under d at level m-1, to *show* $[\forall y S(\alpha)]\phi$ *it remains* to *show* $S(\alpha\phi_1), \cdots, S(\alpha\phi_k)$ and conditions e, if and only if

1) y does not appear in α or , if it does , then $y\theta$ is not a constant and is not equal to $x\theta$ for any other variable symbol x from α;

2) e is a list of inequalities containing exactly those $w \neq t$ from d such that neither w nor t are $y\theta$; and

3) $\phi_1, \cdots, \phi_k$ are exactly those substitutions ϕ_i for which there is either

3a) a constant c such that $y\theta \neq c$ or $c \neq y\theta$ appears in d and ϕ_i acts as ϕ on the other variables in α but maps y to c or

3b) a variable x $\neq$y from α such that $x\theta \neq y\theta$ or $y\theta \neq x\theta$ appears in d and ϕ_i acts as ϕ on the other variables in α but maps y to $x\phi$.

We say that d *implies* G under θ at level m if and only if there is a sequence $G_0, \cdots, G_k$ of goals and a sequence $\theta_0, \cdots, \theta_{k-1}$ of substitutions such that

1) $G_0 = G$,

2) G_k is empty or is all inequalities and $G_k = d$,

3) $\theta = \theta_1 \cdots \theta_{k-1}$

4) for each i = 0... k-1, G_{i+1} is got from G_i via θ_i at level m.

G *succeeds* at level m with substitution θ under conditions d if and only if

1) d implies G under θ at level m,

2) there is no term t such that $t \neq t$ appears in d,

3) there is no variable appearing in d but not in $G\theta$.

G *can flounder* at level m if and only if

there is d and θ such that

1) d implies G under θ at level m,

2) there is no term t such that $t \neq t$ is in d

3) there is a variable in d which is not in $G\theta$

G *can be extended* at level m if and only if there is H and θ such that H is got from G via θ at level m.

G *can flounder immediately* at level m if and only if

1) L is selected from G

2) L is $\forall y S(\alpha)$

3) m > 0

4) $S(\alpha)$ can flounder at level m-1.

G *fails immediately* at level m if and only if

1) L is selected from G

2) either

2.1) L is $t \neq t$ for some term t,

2.2) L is atomic and does not unify with any clause's head

2.3) L is $\forall y S(\vec{x})$, m>0 and $S(\vec{x})$ either finitely fails at level m-1 or if it succeeds with substitution θ under conditions d at level m-1 then $y\theta$ is a constant or is equal to $x\theta$ for another symbol x from $\vec{x}$.

G *finitely fails* at level m if and only if there is a closed bounded set G of unsuccessful m-paths for G which includes G itself.

An m-*path* for G is a sequence $G_0, \cdots, G_k$ of goals and a sequence $\theta_0, \cdots, \theta_{k-1}$ of substitutions such that G_0 is G and each G_{i+1} is got from G_i via θ_i at level m.

A path $G_0, \ldots, G_k$ with $\theta_0, \cdots, \theta_{k-1}$ is *successful* if and only if G_k is empty or all inequalities and contains no variables x which do not appear in $G_0\theta_0 \cdots \theta_{k-1}$.

G is *closed* if and only if if $G_0, \cdots, G_k$ with $\theta_0, \cdots, \theta_{k-1}$ is a path in G then either

1) the path is successful

2) G_k fails immediately at level m, or

3) if there is a goal G_{k+1} and a substitution θ_k of the variables in G_k such that G_{k+1} is got from G_k via θ_k at level m then the extended path is in G.

G is *bounded* if and only if there is a bound on the lengths of paths in G.

EXAMPLE:

It is easy to see that likes(x,y) succeeds at level 0 from the program

popular(y) $\leftarrow \forall x$ likes(x,y)

likes(Mary, Mary) $\leftarrow$

likes(John , Mary) ←

likes(x , Mary) ← $x \neq Mary$, $x \neq John$.

with a substitution θ taking x to x and y to Mary under conditions $x \neq Mary$, $x \neq John$.

If we are trying to show $\forall x$ likes(x,y) then it remains to show x → Mary and x → John under no conditions. This can be done as $x\theta$ = x is not a constant nor equal to $y\theta$ i.e. Mary.

Thus ← likes(Mary , Mary) , likes(John , Mary) can be got from ← $\forall x$ likes(x , y) using θ at level 1.

So popular(Mary) eventually succeeds.

From the following shorter program popular (Mary) finitely fails.

popular(y) ← $\forall x$ likes(x,y)

likes(Mary, Mary) ←

likes(x , Mary) ← $x \neq Mary$, $x \neq John$.

This can be seen because we have a closed bounded set of unsuccessful 1-paths for popular(y) - namely all the sub-paths of the following element.

← popular(y)

← $\forall$ x likes(x,y)

← likes(Mary , Mary) , likes (John , Mary)

← likes(John , Mary)

We can not continue ← $\forall$ x likes (x , y) in any other way, in particular we can not use the fact that likes (x ,y) succeeds with substitution θ taking x to Mary and y to Mary because xθ is a constant.

8.3 Soundness of the QEP

In this section we will establish the easiest result about the QEP. We will also give two conjectures about other results which together would make the new language something worth considering.

First a soundness result.

THEOREM 8.1

If R(α) succeeds with substitution χ then [PROG] $\cup INEQ(C) \vdash \forall R(\alpha\chi)$.

Before giving a proof of this result let us look at two other results which would establish QUILOG as a useful alternative to LP with NAF and which the author believes are true.

CONJECTURE 8.2

If R(α) finitely fails from PROG then $\exists R(\alpha)$ is not a valid deduction from [PROG] $\cup$ INEQ(C).

and

CONJECTURE 8.3

IF NAFPROG is a logic program with NAF and no function symbols then there is a QUILOG program PROG (constructed easily from SYNTH(NAFPROG)) such that if R(α) succeeds under χ from NAFPROG then R(α) succeeds under χ from PROG and if R(α) finitely fails from NAFPROG then it finitely fails from PROG.

Now let us prove the soundness result.

proof:

Suppose that M is a model of [PROG] $\cup INEQ(C)$. We show by induction on m that if d implies G under θ at level m then M is a model of $\forall x(\wedge d \rightarrow \wedge G\theta)$. Since the empty list of conditions implies R(x) under χ at some level m then M $\models \forall x(R(x\chi))$ as required.

Suppose that ϕ is an M-substitution of x such that M $\models \wedge d\phi$. We must show that M is a model of $\wedge G\theta\phi$.

There is a sequence $G_0, \cdots, G_k$ of goals and $\theta_0, \cdots, \theta_{k-1}$ of substitutions such that G_0 is G, θ is $\theta_0 \cdots \theta_{k-1}$, G_k is d and each G_{i+1} is got from G_i via θ_i at level m. We show that for each i, M $\models \wedge G_i\theta_i \cdots \theta_{k-1}\phi$. This is true for i=k and we now work back to i=0 which will give us our result.

Suppose that M is a model of $\wedge G_{i+1}\theta_{i+1} \cdots \theta_{k-1}\phi$, that G_i is $\leftarrow L_1, \cdots, L_p$ and that L_j is selected. There are three cases.

case 1 L_j is an inequality.

Suppose that L_j is $a \neq b$, G_{i+1} is $\leftarrow L_1, \cdots, L_{j-1}, L_{j+1}, \cdots, L_p$ and θ_i is the identity. We know that M is a model of $\wedge G_{i+1}\theta_{i+1} \cdots \theta_{k-1}\phi$, so if r $\neq$ j then M is a model of $L_r\theta_i\theta_{i+1} \cdots \theta_{k-1}\phi$. If r=j then, since M $\models INEQ(C)$, we have M $\models a \neq b\theta_i \cdots \theta_{k-1}\phi$, giving us M $\models \wedge G_i\theta_i \cdots \theta_{k-1}\phi$ as required.

case 2 L_j is a literal.

Suppose that L_j is R(x), that R(x) from G_i unifies with the head A from clause $A \leftarrow A_1, \cdots, A_q$ with mgus θ_i, ψ. Thus G_{i+1} is $\leftarrow L_1\theta_i, \cdots, L_{j-1}\theta_i, A_1\psi, \cdots, A_q\psi, L_{j+1}\theta_i, \cdots, L_p\theta_i$. So, by our inductive hypothesis, M is a model for $(L_1\theta_i \wedge \cdots \wedge L_{j-1}\theta_i \wedge A_1\psi \wedge \cdots \wedge A_q\psi \wedge L_{j+1}\theta_i \wedge \cdots \wedge L_p\theta_i)\theta_{i+1} \cdots \theta_{k-1}\phi$.

Since M is a model of $L_s\theta_i \cdots \theta_{k-1}\phi$ for each s different from j, to show that M is a model of $\wedge G_i\theta_i \cdots \theta_{k-1}\phi$ we need only show that it is a model of $L_j\theta_i \cdots \theta_{k-1}\phi$. However $L_j\theta_i$ equals $A\psi$ so we are done when we show that M is a model of $A\psi\theta_{i+1} \cdots \theta_{k-1}\phi$.

Now M is a model of [PROG] so it is a model of $\forall(\wedge_{i=1}^{j-1}A_i \rightarrow A)$. We also know that it is a model of $\wedge_{i=1}^{j-1}A_i\psi\theta_{i+1} \cdots \theta_{k-1}\phi$. Hence we have M $\models A\psi\theta_{i+1} \cdots \theta_{k-1}\phi$ as required.

case 3 - L_j is universal.

Suppose that L_j is $\forall y S(\pi)$. So m>0, $S(\pi)$ succeeds at level m-1 with substitution η under conditions f, with $S(\pi)$ succeeding with η under f, to show $[\forall y S(\pi)]\ \eta$ it remains to show e and $S(\pi\phi_1)$, ... , $S(\pi\phi_t)$, θ_t is an mgg of η to G_t and G_{t+1} is $\leftarrow(L_1\theta_t), \cdots, (L_{j-1}\theta_t), e, S(\pi\phi_1), \cdots, S(\pi\phi_k), (L_{j+1}\theta_t), \cdots, (L_p\theta_t)$.

By our inductive hypothesis on i, M is a model of $L_r\theta_i\theta_{i+1} \cdots \theta_{k-1}\phi$ for each r different from j. We must show that M is a model for $L_j\theta_i \cdots \theta_{k-1}\phi$. We also know that M is a model of $(\wedge e)\theta_{i+1} \cdots \theta_{k-1}\phi$ and of each $S(\pi)\ \phi_r\theta_{i+1} \cdots \theta_{k-1}\phi$.

By our inductive hypothesis on m, M is a model of $\forall z\tau(\wedge f \rightarrow S(\pi\eta))$ where z is $y\eta$ and τ are the distinct variables in $\pi\eta$ apart from z. We know that there are not any other variables in f, that $y\eta$ is in fact a variable and that it, z, is not equal to $x\eta$ for any x different from y in π.

Suppose that $L_j\theta_i \cdots \theta_{k-1}$ is $\forall w[S(\pi)(\tau\backslash x\theta_i \cdots \theta_{k-1}, y\backslash w)]$. So τ are the variables in π apart from y and w does not appear in $x\theta_i \cdots \theta_{k-1}$. To show that M is a model of $L_j\theta_i \cdots \theta_{k-1}\phi$, we must show that, for each element c of M, M $\models S(\pi)(\tau\backslash x\theta_i \cdots \theta_{k-1}\phi, y\backslash c)$. So suppose that c is an element of M.

We define an M-substitution ϵ of z , τ as follows. Put $z\epsilon = c$. If $z_h = x_g\eta$, put $z_h\epsilon = x_g\theta_i \cdots \theta_{k-1}\phi$. We must check that this is well-defined. If $z_h = x_g\eta = x_{g'}\eta$, then, since

θ_l is an mgg of η, $x_g\theta_l = x_{g'}\theta_l$, so $x_g\theta_l \cdots \theta_{k-1}\phi = x_{g'}\theta_l \cdots \theta_{k-1}\phi$. Thus ϵ is well-defined and clearly $\alpha(x \backslash x\theta_l \cdots \theta_{k-1}\phi, y \backslash c) = \alpha\eta\epsilon$.

There are two cases.

case 2a - M is a model of $\bigwedge f\eta$.

Then M is a model of $S(\alpha)\ \eta\epsilon$ as required.

case 2b - M is not a model of $\bigwedge f\epsilon$.

Let us suppose that M is not a model of $w \neq t\ \epsilon$. There are three cases.

case 2b1 - neither w nor t are z.

In this case $w \neq t$ is in e. But M is a model of $\bigwedge e\theta_{l+1} \cdots \theta_{k-1}\phi$ so M is actually a model of $w \neq t\ \epsilon$. This case does not arise.

case 2b2 - one, say w, is z and t is the constant a.

Since M is not a model of $w \neq t\ \epsilon$ it is a model of $z = a\ \epsilon$. So a=c (only the same constant symbols have the same interpretation in M). Now we know that one of the ϕ_s takes y to c because one inequality , $w \neq t$, in f is $y\eta \neq c$. M is a model of $S(\alpha\phi_s)$ $\theta_{l+1} \cdots \theta_{k-1}\phi$ but this is just $S(\alpha)\ (x \backslash x\theta_l \cdots \theta_{k-1}\phi, y \backslash c)$.

case 2b3 - one, say w, is z and t is $x\eta$.

So M is a model of $y\eta\epsilon = x\eta\epsilon$. Thus $x\theta_l \cdots \theta_{k-1}\phi$ is c. But we know that there is a ϕ_s taking y to $x\theta_l$ such that M is a model of $S(\alpha\phi_s)\ \theta_{l+1} \cdots \theta_{k-1}\phi$. Thus M is a model of $S(\alpha)\ (x \backslash x\theta_l \cdots \theta_{k-1}\phi, y \backslash c)$ as required.

Q.E.D.

Chapter 9

Conclusions

In this concluding chapter, we restate the objective of this thesis and its motivation, assess how successful the thesis was in attaining this objective, point out some of the interesting and useful by-products of our research and mention some related open problems.

9.1 Review of aims and motivation

We set out to try and find a nice way of extending Datalog to take account of the technology of NAF and to use whatever progress we made on that task to help us assess the expressive power of the LPNAF implementation. Let us just recall why we did this.

We saw that there are two schools of database research - the deductive database and the conventional database schools. One idea which incorporates aspects of both schools is the Datalog language and its implementation using pure logic programming. However, we saw that Datalog was not adequate to express some queries that fixed point languages can handle and we know that logic programmers feel the need to use negation as failure to improve the power of pure logic programming. There is clearly the need for an improvement on Datalog.

The use of negation as failure is, as we saw in chapter one, fraught with problems. The only satisfactory account of its behaviour is the definition of its query evaluation

process and the technical non-intuitive nature of this does not lend it well to the task of explaining the behaviour of logic programs in non-logic-programming terms.

It would be good if we could extend Datalog to a nice query language which clearly represented the technology of NAF. In any case, if that is not possible, there is still the technical question of the power of the LPNAF approach to be answered.

9.2 Account of Results

Let us review the major results which we have established.

In chapter two, we defined a query language DEF which is an extension of Datalog and is nice in the sense that its syntax is quite easily specified (essentially just the addition of conjuncts containing universal quantifications and inequalities to the bodies of Datalog clause) and its semantics are the very intuitive provability semantics (which Datalog also uses).

In chapter three, we were eventually able to prove that DEF is equally as expressible as the most popular fixed point language, FO+LFP, which appears , for example, in [Gur84].

In chapter six, using the results of chapters four and five, we have specified a particular implementation, LPNAF , of the general idea of putting impure logic programs after descriptions of queried structures and shown that it is an implementation of exactly the queries in DEF.

There is one reservation to this last result. We have seen that the actual version of logic programming with negation as failure - the actual selection rule used - is not specified when we say that LPNAF implements each query in DEF. All we know is that there is such a version. In fact, from what we have shown, it can not be ruled out that

the selection rule must be changed when we ask the same query of different structures. It would be better if we could define some class of selection rules which can be used to completely implement the generated program and better still, if, given any DEF query, we could generate a program which could be used with the standard left-first selection rule to implement the query in LPNAF.

The other result in chapter six, that any query implementable in LPNAF is actually a DEF query, needs no apologies.

For good measure, in chapter seven, we have show that the only other claim to similar results (of which the author is aware) is unfounded.

Finally, in chapter eight, we tackle the problem of the non-declarative nature of LPNAF. We suggest that logic programming has the capability of being extended to handle the necessary increased expressive power and to, at the same time, still remain declarative.

9.3 Further Work for the Database theorist

It is worth pointing out some of the ways in which results from this thesis can be used for other purposes. First, let us see what there is for people interested in database questions.

One constructive thing that can be done is for the methods of translation between the different languages FO+LFP, $\overline{LFP}$ and LPNAF to be extracted from the proofs of their equivalence. Each of the appropriate proofs is done constructively although this fact was not emphasised. This holds for the proof that FO+LFP is not more expressive than $\overline{LFP}$ (despite the fact that it does not look as though it does) because Immerman's result, which is used in the proof, is proven constructively.

With all these transformations readily available, one can find an implementation - whether in LPNAF , QUILOG or some recursive technique - for any query expressed recursively, as a fixed point, as a logical consequence or as a logic program.

By extracting the transformation out of Immerman's result showing that FO+LFP expresses exactly the polynomial time computable queries on an ordered structure we can extend the flexibility of our query answering abilities a lot further too.

We looked at the question of complexity of queries in chapter seven. It was made clear there that DEF stops short of capturing all the polynomial time computable queries. Immerman considered extending the equivalent language FO+LFP to capture this class. Perhaps there is a provability language and method of implementation which gets there. However, the interested researcher is recommended to consult [Gur84] first.

To put such implementations to good use, it would be helpful for the researcher to know which are good ways of implementing queries and which are not. LPNAF might be excellent at finding one correct answer to a query but the user who wants to know them all had better use some bottom-up whole relation implementation.

Theres a lot of practical work to be done classifying the uses that query answering techniques will be put to and assessing the relative merits of the different ways of answering them or coming up with better ways. The questions of expressive power, covered in this thesis are only the beginning.

9.4 Further Work for the Logic Programmer

While the database researcher is looking at the rest of the thesis the logic programmer who is not necessarily interested in database questions will be interested in chapters four and five of the thesis. If function symbols can be incorporated into the

results of the these chapters then lots of important results can be proven or almost proven.

In chapter four we have shown how to define a maximal selection rule for a given program using negation as failure. Shepherdson, in [She84], also defines such a rule. In some cases, for example if there is a certain ordering on the atomic propositions of the language, it might be easy to predict the behaviour of such a rule. Such predictions will be useful for making the practice of logic programming easier. A computer which, when using one of those programs, could select the right literal, would get all the answers that logic programming is capable of getting. With the results of chapter five, we know a lot about these answers.

It looks fairly straight forward for the SYNTH construction of chapter five to be generalised to the functional case. The correctness results should follow. In the completeness result, the covering condition can be relaxed a lot - it was only made so strict because that made the proof easy and it did not need to be less strict to get the result in chapter six. However, there might have to be conditions introduced on the appearances of function symbols in the program.

Even the proof in this thesis is a significant advance on the unsatisfactory Clark results so it should not be hard to get useful results.

The other results of interest to the logic programmer are those concerning QUILOG. There is still a lot of work to do on QUILOG - prove that it is correct, that it is complete in some cases, generalise it and these results to the function case and actually implement it on a computer.

There is another possibility for developing a powerful declarative language which is suggested by work in this thesis. By approaching logic programming with negation as failure from a fixed point point of view - by generalising the relationship between LPNAF

and *LFP* - we might be able to define a declarative programming language based on inductive definitions. When the completeness result of chapter five is generalised to the function case, what might be needed is exactly the kind of ordering on atomic propositions which an inductive definition involves.

Good luck with these problems.

Appendix

Technical result 1

If **L** is a signature, P is an n-ary symbol not in **L**, D is an **L**-structure, $\phi = \phi(x)$ an **L**$\cup\{P\}$-formula, P appears only positively in ϕ and $\sigma \leqslant \rho$ are n-ary relations on D then for all a from D,

$D \models \phi(P\backslash\sigma, a)$ implies $D \models \phi(P\backslash\rho, a)$.

proof:

By induction on the construction of **L**$\cup\{P\}$-formula ϕ we will prove the following: if $\sigma \leqslant \rho$ then

P only positive in ϕ implies $D \models \phi(P\backslash\sigma, a) \Rightarrow D \models \phi(P\backslash\rho, a)$ and

P only negative in ϕ implies $D \models \neg\phi(P\backslash\sigma, a) \Rightarrow D \models \neg\phi(P\backslash\rho, a)$.

If P does not appear in ϕ, then clearly both implications hold.

If ϕ is $P(x)$ then P appears only positively and we find that $D \models \phi(P\backslash\sigma, a) \Rightarrow a(x) \in \sigma \Rightarrow a(x) \in \rho \Rightarrow D \models \phi(P\backslash\rho, a)$.

If ϕ is $\theta \wedge \chi$ and P appears only positively in ϕ then it appears only positively in θ and χ. So $D \models \phi(P\backslash\sigma, a) \Rightarrow D \models \theta(P\backslash\sigma, a)$ and $D \models \chi(P\backslash\sigma, a) \Rightarrow D \models \theta(P\backslash\rho, a)$ and $D \models \chi(P\backslash\rho, a) \Rightarrow D \models \phi(P\backslash\rho, a)$. Similarly for ϕ equalling $\theta \vee \chi$, $\exists x\,\theta$ and $\forall x\,\theta$.

If P appears only negatively in such ϕ it is equally easy to get the corresponding result.

If ϕ is $\neg\theta$ and P appears only positively in ϕ then P appears only negatively in θ. So $D \models \phi(P\backslash\sigma, a) \Rightarrow D \models \neg\theta(P\backslash\sigma, a) \Rightarrow D \models \neg\theta(P\backslash\rho, a) \Rightarrow D \models \phi(P\backslash\rho, a)$.

If P appears only negatively in $\neg\theta$ then its equally easy to get the desired result.

Q.E.D.

Technical result 2

Suppose that L and P are signatures, for each i=1...m, $\phi_i(x_i)$ is an L∪P-formula in which each P from P appears only positively, $\bar{P}=(P_1, \cdots, P_m)$ is a list of the distinct symbols from P which appear in ϕ, each P_i is m_i-ary, $\bar{m}=(m_1, \cdots, m_m)$, D is a finite L-structure and $F:REL_{\bar{m}}(D)\rightarrow REL_{\bar{m}}(D)$ is given by $\bar{a}\in[F(\bar{p})]_i \iff D\models\phi_i(\bar{P}\backslash\bar{p},\bar{x}_i\backslash\bar{a})$, then F is monotone on $REL_{\bar{m}}(D)$.

proof:

Suppose that $\bar{\sigma}\leqslant\bar{p}$. Then for all i, if $\bar{a}\in[F(\bar{\sigma})]_i$ then $D\models\phi_i(\bar{P}\backslash\bar{\sigma},\bar{x}_i\backslash\bar{a})$. Using TR1 m times we get $D\models\phi_i(\bar{P}\backslash\bar{p},\bar{x}_i\backslash\bar{a})$. So $\bar{a}\in[F(\bar{p})]_i$.

Thus $[F(\bar{\sigma})]_i \leqslant_{m_i} [F(\bar{p})]_i$.

Q.E.D.

Technical result 3

Notation as in section 3.5.

For all $\bar{m}$-ary relations τ on D, we define an M+e-ary relation $coded(\tau)$, by $(\bar{d},\bar{b})\in coded(\tau)$ if and only if

there is an i such that

C1) $D\models code_i(\bar{b})$

C2) $\bar{d}=(\bar{a},\bar{c})$ and $\bar{a}$ is m_i-ary and

C3) $\alpha \in \tau_i$.

Then for all i, if $D \vdash \phi_i(P \backslash coded(\tau), x_i \backslash \alpha)$ then $D \vdash \psi_i(P \backslash \tau, x_i \backslash \alpha)$.

proof :

By induction on the construction of ψ_i.

case 1 : no P_ks appear in ψ_i.

Then P does not appear in ϕ_i which is the same as ψ_i and we have our result.

case 2 : $\psi_i = P_k(\alpha)$.

In this case $\phi_i = \exists x y (code_k^i \wedge (x_1 = u_1) \wedge \cdots \wedge (x_k = u_k) \wedge P(x,y))$.

So if $D \vdash \phi_i(P \backslash_{\wedge coded(\tau)}, x_i \backslash \alpha)$ then there is σ and $\bar{g}$ such that $D \vdash code_k^i(y \backslash \bar{g})$, $D \vdash (c_1 = u_1(\alpha)) \wedge \cdots \wedge (c_{m_k} = u_{m_k}(\alpha))$ and $D \vdash P(\sigma, \bar{g}) (P \backslash coded(\tau))$.

So $(\sigma, \bar{g}) \in coded(\tau)$. which implies that there is a j such that $D \vdash code_j^i(\bar{g})$, $\sigma = (\overline{c1}, \overline{c2})$, $\overline{c1}$ is m_j-ary and $\overline{c1} \in \tau_j$. But then j=k as $\bar{g}$ can not satisfy $code_k^i$ and $code_j^i$ in D for different j and k.

Thus $\overline{c1} = \alpha(\alpha)$ and $D \vdash P_k(\alpha)(P \backslash \tau, x_i \backslash \alpha)$.

case 3 : ψ_i is more complicated.

Its easy to get the result assuming the inductive hypothesis on the maximal proper subformulas of ψ_i.

Q.E.D.

Technical result 4

Using the notation of section 3.5 and of technical result 3.

For all finite L-structures D, for all $\bar{d}$ from D^{m_l}, if $D \models s_l^*(\bar{d})$ then $D \models \overline{LFP}(s;P_l)(\bar{d})$.

proof:

Suppose that $\chi \rightarrow P(x,y)$ prescribes f on D.

First of all we show that for any τ, $f(coded(\tau)) \leqslant coded(F(\tau))$.

So suppose that $(\bar{c},\bar{b}) \in f(coded(\tau))$. Then $D \models \chi(P \backslash coded(\tau), x \backslash \bar{c}, y \backslash \bar{b})$. So there is an i such that $D \models \chi_i(P \backslash coded(\tau), x \backslash \bar{c}, y \backslash \bar{b})$.

Let us suppose that $\bar{c} = (\bar{c}1, \bar{c}2)$ with $\bar{c}1$ m_i-ary. By definition of χ_i we have $D \models code_i^l(y \backslash \bar{b})$ and $D \models \phi_i(P \backslash coded(\tau), x_i \backslash \bar{c}1)$. By technical result 3, we have $D \models \psi_i(P \backslash \tau, x_i \backslash \bar{a})$ and hence $\bar{a} \in [F(\tau)]_i$ from which the result follows.

Now back to the main proof.

We have f(∅) ⩽ coded(F(∅)). $f^2(\emptyset) \leqslant f(coded(F(\emptyset)))$. ⩽ coded (F (F (∅))). etc. From this we can deduce that $lfp(f) \leqslant coded(\overline{lfp}(F))$.

If $D \models s_l^*(\bar{d})$ then there is $\bar{a}$ and b such that $D \models LFP_{P,xy}(\chi)(\bar{d},\bar{a},b,\cdots,b)$. Thus $(\bar{d},\bar{a},b,\cdots,b)$ is in $lfp(f)$ and from what we have just established it is also in $coded(\overline{lfp}(F))$.

By definition, there is i such that $D \models code_i^l(b,\cdots,b)$ so that i=l, and $\bar{d} \in [\overline{lfp}(F)]_l$.

Thus $D \models \overline{LFP}(s;P_l)(\bar{d})$ as required.

Q.E.D.

Technical result 5

Notation as in technical result 6.

if $D \models \psi_i(P \backslash \tau, x_i \backslash \bar{d})$ then $D \models \phi_i(P \backslash combi(\tau, g, h), x_i \backslash \bar{d})$.

proof:

By induction on the construction of ψ_i.

Case 1 : ψ_i contains no P_ks.

Then $\psi_i = \phi_i$ and both conditions are $D \models \psi_i(x_i \backslash \bar{d})$.

Case 2 : ψ_i is $P_k(\alpha)$.

Suppose that $D \models P_k(\alpha)(P \backslash \tau, x_i \backslash \bar{d})$. Then $\alpha(\bar{d}) \in \tau_k$, and if we put $\bar{b} = code(g, h, \bar{d})$ then $D \models code_k^i(\bar{b})$ and $(\alpha(\bar{d}), \bar{c}, \bar{b}) \in combi(\tau, g, h)$.

Since this latter condition implies $D \models P(\alpha(\bar{d}), \bar{c}, \bar{b})(P \backslash combi(\tau, g, h))$, we have the result that $D \models \phi_i(P \backslash combi(\tau, g, h), x_i \backslash \bar{d})$.

Case 3 : ψ_i is more complicated.

This is easy using the inductive hypothesis.

Q.E.D.

Technical result 6

Notation as in section 3.5.

For all finite non-trivial L-structures D, for all $\bar{d}$ from D^{m_i}, if $D \models LFP(s; P_i)(\bar{d})$ then $D \models s_i^{\tau}(\bar{d})$.

proof:

Say s prescribes F on D , with its lfp being π. Then

$s_i^* = \exists x_{m_i+1} \cdots x_m y_0 LFP_{P\pi \wedge y}(\chi)(x, y_0, \cdots, y_0)$. Suppose tha

with lfp σ.

So we know $\bar{d} \in [\pi]_{P_i}$. Choose any two different elements g and

Notation. Suppose that b_j is the jth bit in the binary representation

then code(g , h , i) is the e-tuple $(c_1, \cdots, c_e)$ from D^e with $c_j = g$ if $b_j =$

$b_j = 1$.

If p an π-ary relation on the domain of D, then $combi(p, e, g)$ is the m+$\log_2 k$ + relation given by

$(a, c, b) \in combi(p, e, g)$

if and only if

there is i such that

1) $a \in \rho_i$

2) c is any (m-m_i)-tuple

3) b = code(g , h ,i).

Say $\bar{d} \in [F^m(\varnothing)]_{P_i}$. So $(\bar{d}, g, \cdots, g, g, \cdots, g) \in combi(F^m(\varnothing), e, g)$.

Now $combi(F(p) g, h) \leqslant f(combi(p, g, h)$ as we now show. Suppose that $(a, c, b) \in combi(F(p), g, h)$. Then 1) $a \in [F(p)]_i$ and 2) $b = code(g, h, i)$.

But we claim that $D \models \chi_i(combi(p, g, h), x \backslash a \wedge c, y \backslash b)$. First $D \models code_i(b)$ as $b = code(g, h, i)$. Second, $D \models (a_1 = a_1) \wedge \cdots \wedge (a_{m_i} = a_{m_i})$.

Third, $D \models \phi_i(combi(p, g, h), x_i \backslash a)$ by TR5 as we know that $D \models \psi_i(p, x_i \backslash a)$.

So $D \models \chi(combi, x \backslash a \wedge c, y \backslash b)$ so $(a, c, b) \in f(combi(p, g, h))$.

Back to our main proof. We have $(d, g, \cdots, g, g, \cdots, g) \in combi(F^m(\varnothing), g, h)$

$\leqslant f(combi(F^{m-1}(\varnothing), g, h)) \leqslant f^2(combi(F^{m-1}(\varnothing), g, h))$

$\cdots$

$\leqslant f^m(combi(\varnothing, g, h))$

$= f^m(\varnothing)$

$\leqslant \sigma$.

So $(d, g, \cdots, g) \in \sigma$. So $D \models s_i^+(d)$.

Q.E.D.

Technical result 7

Notation as in section 3.5.

For all finite L-structures D, if s prescribes G on D and $d \in [G(\varnothing)]_{P_i}$ then $D \models s_i^+(d)$.

proof:

Suppose that $d \in [G(\varnothing)]_{P_i}$. So $D \models \psi_i(P \backslash \varnothing, x_i \backslash d)$.

Since $\varnothing = coded(\varnothing)$, TR3 implies that $D \models \phi_i(P \backslash \varnothing, x_i \backslash d)$.

Now choose any element g from D and let $g = (g, \cdots, g) \in D^s$. Thus $D \models code_i^s(g)$. Choose any $(M - m_i)$-tuple c from D. Thus $D \models \chi_i(P \backslash \varnothing, x \backslash (d, c), y \backslash g)$ and hence $D \models \chi(P \backslash \varnothing, x \backslash (d, c), y \backslash g)$.

Suppose that f is the operator prescribed by $\chi \rightarrow P(x, y)$ on D. So $(d, c, g) \in f(\varnothing)$ and (d, c, g) is in the least fixed point of f.

Thus D⊨$LFP_{P\pi\rho}(\chi)(\bar{d},\sigma,\varepsilon)$ and we have our desired result that D⊨$s_i'(\bar{d})$.

Q.E.D.

Technical result 8

Notation as in 3.5.

If $[\psi_1 \rightarrow P_1(\tau_1), \cdots, \psi_m \rightarrow P_m(\tau_m)]$ prescribes G on D, n = 1 ... or m and $\bar{d}$ is an m_n-tuple of elements of D then

D⊨$app(\psi_n)(P\backslash\rho,\tau_n\backslash\bar{d})$

if and only if

either D⊨$\psi_n(P\backslash\rho,\tau_n\backslash\bar{d})$ or D⊨$\psi_n(P\backslash G(\rho),\tau_n\backslash\bar{d})$.

proof:

By induction on the construction of ψ_n.

Case 1 : ψ_n does not contain any P_qs.

The "only-if" direction is easy in this case as $app(\psi_n)=\psi_n$. The "if" direction is almost as easy when it is realised that D⊨$\psi_n(P\backslash G(\rho),\tau_n\backslash\bar{d})$ is true if and only if D⊨$\psi_n(P\backslash\rho,\tau_n\backslash\bar{d})$ because no symbols from P appear in ψ_n.

Case 2 : $\psi_n=P_q(\alpha)$.

So $app(\psi_n)=\psi_q(\tau_q\backslash\alpha) \vee P_q(\alpha)$.

The only if direction.

Suppose that D⊨$app(\psi_n)(P\backslash\rho,\tau_n\backslash\bar{d})$. If D⊨$P_q(\alpha)(P\backslash\rho,\tau_n\backslash\bar{d})$ we have our result so suppose that D⊨$\psi_q(\tau_q\backslash\alpha)(P\backslash\rho,\tau_n\backslash\bar{d})$ instead.

Then $D \vdash \psi_q(P \backslash p, x_q \backslash a(\bar{d}))$. Thus $a(\bar{d}) \in [G(p)]_{P_q}$, and $D \vdash P_q(a)(P \backslash G(p), x_n \backslash \bar{d})$ giving us the result that $D \vdash \psi_n(P \backslash G(p), x_n \backslash \bar{d})$.

The if direction.

Just reverse the argument above.

Case 3 : ψ_n is more complicated.

The proof in this case is an easy use of the inductive hypothesis.

Q.E.D.

Technical result 9

Notation as in section 3.5.

If s prescribes G on D and app(s) prescribes H on d then,
for all m-ary relations p on D, $H(p) = G^2(p) \cup G(p)$.

proof:

Suppose that $s = [\psi_1 \rightarrow P_1(x_1), \cdots, \psi_m \rightarrow P_m(x_m)]$.

(inclusion of $H(p)$ in $G^2(p) \cup G(p)$)

Suppose that $\bar{d} \in [H(p)]_{P_n}]$ so that $D \vdash app(\psi_n)(P \backslash p, x \backslash \bar{d})$. By TR8, we have either $D \vdash \psi_n(P \backslash p, x_n \backslash \bar{d})$ or $D \vdash \psi_n(P \backslash G(p), x_n \backslash \bar{d})$.

In the first case, $\bar{d} \in [G(p)]_{P_n}$ and in the second case $\bar{d} \in [G(G(p))]_{P_n}$ as required.

(other inclusion)

Suppose that $\bar{d} \in G^2(p) \cup G(p)$. So either $D \vdash \psi_n(P \backslash p, x_n \backslash \bar{d})$ or $D \vdash \psi_n(P \backslash G(p), x_n \backslash \bar{d})$ and

again by TR5.3, $D \vdash app(\psi_n \ulcorner P \urcorner \backslash p, x_n \backslash \vec{d})$. Thus $\vec{d} \in [H(p)]_{P_n}$ as required.

Q.E.D.

Technical result 10

If r prescribes F on D and r^* prescribes G on D then F and G have the same least fixed point.

proof:

We show by induction that for all l, if r prescribes F on D and $app^l(r)$ prescribes G on D then F and G have the same least fixed point.

This is true for 0 when $app^0(r) = r$.

Now assume that it is true for l, that r prescribes F on D, that $s = app^l(r)$ prescribes G on D and that $app^{l+1}(r) = app(s)$ prescribes H on D. By our inductive hypothesis F and G have the same least fixed point. We must show that H shares it too.

$lfp(H) \leqslant lfp(G)$

We show by induction on n that $H^n(\varnothing) \subset lfp(G)$ so that for large enough n we have our result.

When n=0 the result is obvious. Now suppose that the result holds for n. Then $H^{n+1}(\varnothing) = H(H^n(\varnothing)) \leqslant H(lfp(G))$ as H is monotone. But this is $G^2(lfp(G)) \cup G(lfp(G))$ by TR9. Because $lfp(G)$ is a fixed point of G we have our result.

$lfp(G) \leqslant lfp(H)$.

$\varnothing \leqslant lfp(H)$.

$G^{n+1}(\varnothing) = G(G^n(\varnothing)) \leqslant G(lfp(H)) \leqslant G(lfp(H)) \cup G^2(lfp(H))$

$= H(lfp(H)) = lfp(H).$

Q.E.D.

Technical result 11

If D is trivial and r prescribes F on D then $F^m(\varnothing)$ is the least fixed point of F in D.

proof:

Suppose that the single element of D is b.

Define $sz(\rho)$ to be the size of the set { i=1...m | ρ_i is the full m_i-ary relation on D }. We show that if $\rho < \sigma$ then $sz(\rho) < sz(\sigma)$. Its obvious that $sz(\rho)$ is less than or equal to $sz(\sigma)$ but there is i such that $\rho_i < \sigma_i$. There are two cases. If $m_i = 0$ then σ_i is true, - the full 0-ary relation on D - while ρ_i is false. So $sz(\rho) < sz(\sigma)$. If m_i is greater than zero then $bbar = (b, \cdots, b)$ from D^{m_i} must be the m_i-tuple in σ_i but not in ρ_i. So σ_i is the full m_i-ary relation on D while ρ_i is not. In both cases we have $sz(\rho_i) < sz(\sigma_i)$.

Suppose that $F^m(\varnothing)$ is not the least fixed point of F. Then we have a strictly increasing chain $\varnothing < F(\varnothing) < \cdots < F^m(\varnothing) < F^{m+1}(\varnothing)$. By the above argument we have a strictly increasing chain of sizes, $sz(\varnothing) < sz(F(\varnothing)) < \cdots < sz(F^m(\varnothing)) < sz(F^{m+1}(\varnothing))$. Thus the last size must be at least m+1. This is impossible given the definition of size.

Q.E.D.

Technical result 12

Suppose that C is a finite set of constant symbols, M is a model of INEQ(C), n is a positive integer, $\bar{m}$ is an n-tuple of elements from M, $\bar{v}$ is an n-tuple of C-terms and $\bar{m}$ can not be got as an M-substitution of $\bar{v}$. Then either

1) there is an i such that v_i is a constant a but m_i is not the interpretation of a in M or

2) there is i,j such that $v_i=v_j$ is a variable symbol but m_i and m_j are different elements of M.

proof:

Suppose that neither 1 nor 2 hold. We define an M-substitution α of $\bar{v}$ as follows. For each variable symbol x in $\bar{v}$, if x appears as v_i put $x\alpha=m_i$.

We must show that this is well-defined. Suppose that $v_i=x=v_j$. Since 2 does not hold m_i and m_j are the same element of M as required.

In fact $\bar{v}\alpha=\bar{m}$. For if v_i is a constant then $v_i\alpha$ is the interpretation of a in M which is also m_i since 1 does not hold. If v_i is a variable symbol then $v_i\alpha=m_i$ by definition of α.

This is a contradiction to the premise that $\bar{m}$ can not be got as an M-substitution of $\bar{v}$.

Q.E.D.

Technical result 13

If the existential player does not have a winning strategy for playing the n-game for $P(\bar{v})$ then the universal player does.

proof:

We describe how the universal player plays so that at every stage the existential player does not have a winning strategy from then on.

This is so at the start.

Suppose that it holds at some stage, there are two cases.

case 1 : it is the existential player's turn

case 1a : she can play to put herself in a position so that she has a winning strategy from then on. This can not arise as then she would have a winning strategy from now on.

case 1b : she can make no such move. So then our condition holds at the next stage of play.

case 2: it is the universal player's turn

case 2a: every choice he makes puts the existential player in a winning position. This can not arise because then she has a winning strategy now.

case 2b: there is a choice so that afterwards the existential player does not have a winning strategy. We get the universal player to make any such choice.

Q.E.D.

Technical result 14

Suppose that M is an atomic formula, PROG is a program and θ a substitution of the variables in M. Then, if M finitely fails then Mθ does too.

proof:

Suppose that M finitely fails at level n. We play the n-game for $M\theta$ as the existential player against Abelard and we get *friend* to play as the universal player in the n-game for M against Eloise using a winning strategy.

At the start the current goal in our game , $\leftarrow M\theta$, is a substitution of the current goal, $\leftarrow$ M, in the other game. We keep this relationship holding.

So suppose that it holds at some stage. Say, in our game, the goal is $\leftarrow L_1\phi, \cdots, L_p\phi$ while it is $\leftarrow L_1, \cdots, L_p$ in the other game. Since Eloise is playing to win that goal will not be empty so neither is ours. She chooses a literal L_j say. We choose $L_j\phi$.

Case 1 : $L_j\phi$ is positive.

If $L_j\phi$ is positive and does not unify then we win. So suppose that it does. Abelard will choose a unifying clause $B \leftarrow B_1, \cdots, B_q$ with mgus χ,ψ say. Out new goal is $\leftarrow L_1\phi\chi, \cdots, L_{j-1}\phi\chi, B_1\psi, \cdots, B_q\psi, L_{j+1}\phi\chi, \cdots, L_p\phi\chi$. Now since $L_j\phi\chi = B\psi$, L_j and B unify. So Eloise has not won her game yet. We get *friend* to choose the clause $B \leftarrow B_1, \cdots, B_q$ too. Say that the mgus are χ',ψ'. So the new goal there is $G = \leftarrow L_1\chi', \cdots, L_{j-1}\chi', B_1\psi', \cdots, B_q\psi', L_{j+1}\chi', \cdots, L_p\chi'$. Since χ',ψ' are mgus of L_j and b but $L_j\phi\chi = B\psi$, there is ω such that $\chi'\omega = \phi\chi$ and $\psi'\omega = \psi'$. So $G\omega$ is our goal and the condition is preserved.

Case 2 : $L_j\phi$ is negative.

Suppose that L_j is not($S(\pi)$). If L_j was not ground then Eloise who selected it would lose. This can not happen so L_j must be ground and so equal to $L_j\phi$. Similarly $S(\pi)$ must either succeed or finitely fail at level n-1. If $S(\pi)$ succeeds then we win so suppose that it does not.

Our new gaol is $\leftarrow L_1\phi, \cdots, L_{j-1}\phi, L_{j+1}\phi, \cdots, L_p$ and the other new goal is $\leftarrow L_1, \cdots, L_{j-1}, L_{j+1}, \cdots, L_p$ so the condition is preserved.

Now the other game can not go on forever as then Eloise would lose. However we have seen that it goes on as long as ours does. So our game must stop eventually too. We have seen that the only way that this can happen is by us winning.

Hence we have a winning strategy as required.

Q.E.D.

Bibliography

[AhU79] A. Aho, J. Ullman, Universality of data retrieval languages, in *6th ACM Symposium on Principles of Programming Languages*, (1979), 110-117.

[AjG85] M. Ajtai and Y. Gurevich, Monotone versus positive, manuscript, (Jan 1985).

[ApE82] K. Apt and M. van Emden, Contributions to the theory of logic programming, *Journal of the ACM* 29, (1982), 841-863.

[ABW86] K. Apt, H. Blair and A. Walker, Towards a theory of declarative knowledge, in *Preprints of Workshop on Foundations of Deductive Databases and Logic Programming, Aug 18-22, Washington, DC*, J.Minker, (ed.), (1986).

[Ban78] F. Bancilhon, On the completeness of query languages for relational databases, in *Proceedings of the 7th Symposium on the Mathematical Foundations of Computer Science*, Springer-Verlag, (1978), 112-123.

[BaR86] F. Bancilhon and R. Ramakrishnan, An amateur's introduction to recursive query processing strategies, in *Proceedings ACM Conference on Management of Data, Washington*, (1986), 16-52.

[BGK84] A. Blass, Y. Gurevich and D.C. Kozen, A zero one law for logic with a fixed point operator, Technical report CRL-TR-39-84, Univ. Michigan, (Sep 1984).

[BoG86] E. Boerger and Y. Gurevich, Hierarchy theorems for extensions of first-order logic, manuscript (1986).

[BoJ74] G. Boolos, R. Jeffrey, *Computability and Logic*, Cambridge University Press, London, (1974).

[BYT85] *BYTE*magazine 10, 8(Aug 1985).

[ChH80a] A. Chandra and D. Harel, Computable queries for relational databases, *Journal of Computer System Science* 21, (1980), 156-178.

[ChH80] A. Chandra and D. Harel, Structure and complexity of relational queries, in *21st IEEE Symposium on the Foundations of Computer Science* (1980), 333-347, (also see [ChH82a]).

[ChH82a] A. Chandra and D. Harel, Structure and complexity of relational queries, Journal of Computer System Science 25, (1982), 99-128.

[ChH82b] A. Chandra and D. Harel, Horn clauses and the fixpoint query hierarchy, in *Proceedings of the ACM Symposium on Principles of Database Systems*, (1982), 158-163, (also see [ChH85]).

[ChH85] A. Chandra and D. Harel, Horn-clause queries and generalizations, *Journal of Logic Programming* 1, (1985), 1-15.

[Cha79] C. Chang, On evaluation of queries containing derived relations in a relational data base, in *Advances in Data Base theory, Vol 1*, H. Gallaire, J. Minker and J-M. Nicolas (eds.), Plenum Press, New York, (1981), 235-260.

[ChK73] C. Chang and H. Keisler, *Model Theory*, North-Holland, Amsterdam, (1973).

[Chu41] A. Church, *The Calculi of Lambda Conversion*, Princeton University Press, Princeton, N.J., (1941).

[Cla78] K. Clark, Negation as failure, in *Logic and Data Bases*, H. Gallaire and J. Minker (eds.), Plenum Press, (1978), 293-322.

[CMcC79] K. Clark and F. McCabe, The control facilities of IC-PROLOG, in *Expert*

Systems in the Micro-Electronic Age, D. Michie (ed.), Edinburgh University Press, (1979), 122-149.

[ClM81] W. Clocksin and C. Mellish, *Programming in PROLOG*, Springer-Verlag, Berlin, (1981).

[CCP80] H. Coelho, J. Cotta and L. Pereira, How to solve it in PROLOG, Report, Laboratorio Nacional de Engenharia Civil, Ministerio de Habitacao e Obras Publicas, Lisbon, (1980).

[Cod70] E. Codd, A Relational Model of data for large shared data banks, *Communications of the ACM*, 13, 6 (June 1970), 377-387.

[Cod72] E. Codd, Relational completeness of data base sublanguages, in *Database Systems*, R. Rustin (ed.), Prentice Hall, (1972), 65-98.

[Cod79] E. Codd, Extending the relational data base model to capture more meaning, *ACM Trans. Database Systems* 4, 4 (Dec 1979), 397-434.

[CKRP73] A. Colmeraurer, H. Kanoui, P. Roussel and R. Pasero, Un systeme de communication homme-machine en francais, Groupe de research en Intelligence Artificielle, Universite d'Aix-Marseille, (1973).

[Dah82] V. Dahl, On database systems development through logic, *ACM Trans Database Systems* 7, 1(1982), 102-123.

[Dar84] J. Darlington, Functional programming, in *Distributed Computing*, F. Chambers, D. Duce and G. Jones (eds.), APIC Studies in Data Processing No. 20, Academic Press, (1984).

[Dat77] C. Date, *An introduction to database systems*, Addison-Wesley, (1977).

[Dat81] C. Date, *An introduction to database systems, 3rd edition*, Addison-Wesley, (1981).

[DaW83] M. Davis and E. Weyuker, *Computability, Complexity and Languages*, Academic press, (1983).

[Del80] C. Delobel, An overview of the relational data theory, *Proceedings of I.F.I.P.80*, Elsevier North-Holland, (1980), 413-426.

[Ehr61] A. Ehrenfeucht, An application of games to the completeness problem for formalised theories, *Fund. Math.* 49, (1961), 129-141.

[Emd78] M. van Emden, computation and deductive information retrieval, in *Formal Description of Programming Concepts*, E.J. Neuhold (ed.), North-Holland, (1978), 421-439.

[EmK76] M. van Emden and R. Kowalski, The semantics of predicate logic as a programming language, *Journal of the ACM* 23, 4 (1976), 733-742.

[Fag74] R. Fagin, Generalized first-order spectra and polynomial-time recognizable sets, in *Complexity of Computation*, R. Karp (ed.), *SIAM-AMS Proceedings 7, AMS*, (1974), 43-73.

[Fag75] R. Fagin, Monadic generalized spectrum, *Zeit.f.math.Logic und Grund.d.Math* 21, (1975), 89-96.

[Fla86] T. Flannagan, The consistency of negation as failure, *Journal of Logic Programming*, 2, (1986), 93-114.

[Fra54] R. Fraisse, Sur quelques classifications des systemes de relations, *Publications Sc. d'Universite D'Alger Ser. A, 1 (1954), 35-182.*

[Gai82] H. Gaifman, On local and non-local properties, *Proceedings of the Herbrand Symposium, Logic Coll.81*, Stern (ed.), North-Holland, (1982), 105-135.

[Gai84] H. Gaifman, A closer look at languages with least fixed point operations, second version, extended abstract, (December 1984).

[GaV83] H. Gaifman and M. Vardi, A simple proof that connectivity is not first-order, manuscript, (1983).

[GMSV87] H. Gaifman, H. Mairson, Y. Sagir and M. Vardi, Undecidable optimization problems for database logic programs, manuscript, (1987).

[GaM78] H. Gallaire and J. Minker (eds.), *Logic and Data Bases*, Plenum, (1978).

[GMN81] H. Gallaire, J. Minker and J-M. Nicolas (eds.), *Advances in Data Base Theory, vol 1*, Plenum, (1981).

[GMN84a] H. Gallaire, J. Minker and J-M. Nicolas (eds.), logic and databases: a deductive approach, *Computing Surveys* 16, 2 (June 1984), 153-185.

[GMN84b] H. Gallaire, J. Minker and J-M. Nicolas, *Advances in Database theory, vol 2*, Plenum, (1984).

[GaJ79] M. Garey, D. Johnson, *Computers and Intractability*, W.H. Freeman and company, San Francisco, (1979).

[Gra77] J. Grant, Null values in a relational data base, *Inf. Process. Lett 6, 5 (1977), 156-157.*

[Gre69] C. Green, Theorem proving by resolution as a basis for question-answering systems, in *Machine Intelligence 4*, B. Meltzer and D. Michie (eds.), Elsevier North-Holland, (1969), 183-205.

[Gur83] Y. Gurevich, Algebras of feasible functions, in *24th Symposium on the Foundations of Computer Science, IEEE*,Computer Science Press, (1983).

[Gur84] Y. Gurevich, Toward logic tailored for computational complexity, in *Computation and Proof Theory*, M.M. Richter (ed.), Springer Lecture Notes in Math 1104, (1984), 175-216.

[Gur85] Y. Gurevich, Logic and the challenge of computer science, in *Current trends in Theoretical Computer Science*, E. Boerger (ed.), Computer Science Press, (1985).

[GuS85] Y. Gurevich and S. Shelah, Fixed-point extensions of first-order logic, in *Proceedings of the 26th Symposium on the Foundations of Computer Science*, (May 1985).

[Haj75] Hajek, On logics of discovery, in *Proceedings of the Symposium on the Mathematical Foundations of Computer Science*, LNCS 32, Springer Verlag, (1975), 15-30.

[Har80] D. Harel, Review of Logic and Databases, Gallaire and Minker (eds.), *Computing Reviews* 21, 8 (1980), 367-369.

[HeN84] L. Henschen and S. Naqvi, On compiling queries in recursive first-order databases, *Journal of the ACM* 31, 1 (1984), 47-85.

[Hew72] C. Hewitt, Description and theoretical analysis (using schemata) of PLANNER: a language for proving theorems and manipulating models in a robot, A. I. memo no. 251, MIT project MAC, (1972).

[Hod85] W. Hodges, Reducing first-order logic to Horn logic, manuscript, Queen Mary College, Uni. London, (March 1985).

[Hog84] C. Hogger, *Introduction to Logic Programming*, Academic Press, London, (1984).

[Imm82] N. Immerman, Relational queries computable in polynomial time, in *Proceedings of the 14th ACM Symposium on the Theory of Computing*, (1982), 147-152, (see also [Imm86]).

[Imm83] N. Immerman, Languages which capture complexity classes, *15th ACM Symposium on the Theory of Computing*, (1983), 347-354.

[Imm86] N. Immerman, Relational queries computable in polynomial time, *Information and Control*, 68, (1986), 86-104.

[JLL83] J. Jaffar and J-L. Lassez, Completeness of the NAF rule, in *IJCAI-83, Karlsruhe*, (1983), 500-506.

[JCV84] M. Jarke, J. Clifford and Y. Vassiliou, An optimising PROLOG front-end to a relational query system, in *Proceedings ACM Conference on Management of Data*, (1984), 296-306.

[Kle36] S. Kleene, General recursive functions of natural numbers, *Mathematical Annals*, 112(1936), 727-742.

[Kow74] R. Kowalski, Predicate logic as a programming language, in *IFIP Congress 74 vol 3*, North-Holland, (1974), 569-574.

[Kow78] R. Kowalski, Logic for data description, in [GaM78].

[Kow79] R. Kowalski, *Logic for Problem Solving*, Elsevier North-Holland, (1979).

[Kow81a] R. Kowalski, Logic as a data base language, in *Proceedings of the Advanced Seminar on Theoretical Issues in Data Bases, Cetraro, Italy*, (1981).

[Kow81b] R. Kowalski, PROLOG as a logic programming language, Research report 81/26, Department of Computing, Imperial College of Science and Technology, Uni. London, (1981).

[Kuh67] J. Kuhns, Answering questions by computers - a logical study, Rand memo RM5428PR , Rand Corp., (1967).

[Kuh70] J. Kuhns, Interrogating a relational datafile: remarks on the admissability of input queries, Tech Rept TR-511-PR, Rand Corp., (Nov 1970).

[Kur65] Kurosh, *Lectures on General Algebra*, (Authorised translation by K. Hirsch), Chelsea Publishing Company, New York, (1965).

[Lip79] W. Lipski, On semantic issues connected with incomplete information systems, *ACM Trans. Database systems* 4, 3 (1979), 262-296.

[Liv83] A. Livchak, The relational model for process control, *Automatic Documentation and Mathematical Linguistics* 4, (1983), 27-29.

[Llo83] J. Lloyd, An introduction to Deductive database systems, *Australian Computing Journal 15, 2 (1983), 52-57.*

[Llo84] J. Lloyd, *Foundations of Logic Programming*, Symbolic Computation Series, Springer-Verlag, (1984).

[LlT84] J. Lloyd and R. Topor, Making PROLOG more expressive, *Journal of Logic Programming*, 3 (1984), 225-240.

[LlT85] J. Lloyd and R. Topor, A basis for deductive database systems, *Journal of Logic Programming 2, (1985), 93-109.*

[LlT86] J. Lloyd and R. Topor, A basis for deductive database systems II, *Journal of*

Logic Programming 3 (1986), 55-67.

[Lyn59] R. Lyndon, An interpolation theorem in the predicate calculus, *Pacific Journal of Maths* 9, (1959), 155-164.

[Mai83] D. Maier, *The Theory of Relational Databases*, Computer Science Press, (1983).

[McKS80] D. Mckay and S. Shapiro, Inference with recursive rules, in *Proceedings AAAI Conference on Artificial Intelligence, Stanford, California*, (1980), 151-153.

[McKS81] D. McKay and S. Shapiro, Using active connection graphs for reasoning with recursive rules, in *Proceedings of 7th International Joint Conference on Artificial Intelligence, Vancouver, B.C.*, (1981).

[Min82] J. Minker, On indefinite databases and the closed world assumptions, in *Proceedings of the 6th conference on Automated Deduction*, Springer-Verlag LS number 38, Berlin and New York, (1982), 292-308.

[Mos74] Y. Moschovakis, *Elementary Induction on Abstract Structures*, North-Holland, New York, (1974).

[Nai82] L. Naish, An introduction to MU-PROLOG, Technical Report 82/2, Department of Computer Science, University of Melbourne, (1982).

[NiG78] J-M. Nicolas and H. Gallaire, Database: theory versus interpretation, in [GaM78].

[Nil71] N. Nilsson, *Problem-Solving methods in Artificial Intelligence*, McGraw-Hill, New York, (1971).

[Par78] J. Paredaens, On the expressive power of relational algebra, in *Inf.Process.Lett.*, 7, 2 (Feb 78), 107-111.

[Rei78a] R. Reiter, On closed world databases, in [GaM78], 56-76.

[Rei78b] R. Reiter, Deductive question answering on relational databases, in [GaM78], 149-178.

[Rei78c] R. Reiter, On structuring a first-order database, in *Proceedings of the Second Canadian Society for Computer Science National Conference, Canada,* (1978).

[Rei80] R. Reiter, Equality and domain closure for first-order databases, *Journal of the ACM*, 27, 2 (1980), 235-249.

[Rei84] R. Reiter, Towards a logical reconstruction of relational database theory, in *On Conceptual Modelling*, M. Brodie, J. Mylopoulos and J. Schmidt (eds.), Springer-Verlag, (1984).

[Rob65] J. Robinson, A machine orientated logic based on the resolution principle, in *Journal of the ACM* 12, 1 (Jan 1965), 23-41.

[Rou75] P. Roussel, PROLOG: manuel de reference et d'utilisation, Tech. Rept, Departement d'Informatiques, Universite d'Aix-Marseille, Luminy, Marseille, (1975).

[She84] J. Shepherdson, Negation as failure: a comparison of Clark's completed data base and Reiter's closed world assumption, *Journal of Logic Programming 1, 1 (June 1984), 51-79.*

[She85] J. Shepherdson, Negation as failure II, *Journal of Logic Programming*, 3, (1985), 185-202.

[Tar52] A. Tarski, Some notions and methods on the borderline of algebra and metamathematics, in *Proceedings 1950 Intnl Congress of Math*, AMS, (1952), 705-720.

[Ull82] J. Ullman, *Principles of Database Systems, 2nd edn*, Computer Science Press, (1982).

[Ull85] J. Ullman, Implementation of logical query languages for databases, *ACM Transactions on Database Systems*, 10, 3 (1985), 289-321.

[Var82] M. Vardi, Complexity of relational query languages, *Proceedings of the 14th ACM Symposium on the Theory of Computing*, (1982), 137-146.

[Vas79] Y. Vassiliou, Null values in data base management - a denotational semantics approach, in *Proceedings of ACM-SIGMOD International Conference on the management of Data, Boston*, ACM, New York, (1979), 162-169.

[Vas80] Y. Vassiliou, Functional dependencies and incomplete information, in *Proceedings of the 6th International Conference on Very Large Data Bases*, IEEE, New York, (1980), 260-269.

[WaP81] D. Warren and F. Pereira, An efficient easily adaptable system for interpreting natural language queries, DAI Research Paper number 155, Department of AI, University of Edinburgh, (1981).

[WPP77] D. Warren, L. Pereira and F. Pereira, PROLOG- the language and its implementation compared to LISP, in *Proceedings of SIGART/SIGPLAN Conference on Programming Languages, Rochester, New York*, (1977).

[Zlo79] M. Zloof, Query by example: operations on the transitive closure, RC5526, IBM Research, (1979).